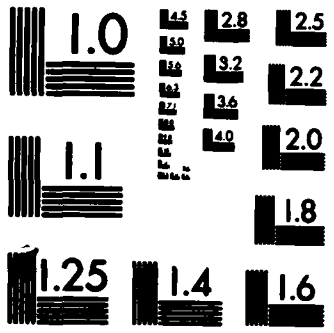
AD-A138 239 PLACING HIDDEN SURFACE REMOVAL WITHIN THE CORE GRAPHICS 1/2
STANDARD: AN EXAMPLE(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. T S WAILES
UNCLASSIFIED 07 DEC 83 AFIT/GCS/MA/83D-9 F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

1

ADA138239

Placing Hidden Surface Removal
Within the CORE Graphics Standard:
An Example

Thesis

AFIT/GCS/MA/~~MA/GCS/~~83D-9 Tom S. Wailes
Capt USAF

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DTIC
ELECTE
FEB 24 1984
S D
B

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DTIC FILE COPY

84 02 21 169

AFIT/GCS/MA/83D-9

Placing Hidden Surface Removal Within the CORE Graphics Standard: An Example

Thesis

AFIT/GCS/MA/~~MA/GCS/~~83D-9 Tom S. Wailes
Capt USAF

DTIC
ELECTE
FEB 24 1984
S D
B

"Original contains color plates: All DTIC reproductions will be in black and white"

Approved for public release; distribution unlimited

MA
AFIT/GCS/~~ENG~~/83D-9

PLACING HIDDEN SURFACE REMOVAL WITHIN THE CORE GRAPHICS STANDARD: AN EXAMPLE

THESIS

Presented to the Facualty of the School of Engineering
of the Air Force Institute of Technology
Air University
in Partial Fulfillment of the
Requirements for the degree of
Master of Science

by
Tom S. Wailes, B.S.
Capt USAF
Graduate Computer Science

December 1983

Approved for public release; distribution unlimited.

Preface

This paper is a testimony to the grace and faithfulness of God. Little did I know, when I embarked on this adventure in the spring of 83, that so many problems could arise on so many pieces of equipment in so short a period of time. God was faithful, and in my many times of dispare, He provided insight and encouragement to get me going again.

A special thanks is due to Dr. John Reising and the many fine people of the crew systems development branch of the Flight Dynamics Laboratory. It was their quest for a microcomputer-based graphics system that spawned this project, and it was their support that made it successful.

Thanks are also in order for the students and faculty of the University of Pennsylvania who shared their Pascal CORE library with the Laboratory. I am greatly in debt to Prof Norman I. Badler who was the principal contact point for information on the design and implementation of the subroutine package.

Perhaps, the one person most responsible for the final outcome of this package is prof Chuck W. Richard Jr. who taught me the basics of computer graphics and has patiently guided me throughout the research. His timely comments and suggestions along with his mathematical expertise made him the ideal advisor for the project.

Finally, I wish to honor the many sacrifices made by

my wife of six and a half years. During this long graduate school experience, she has had to take a second seat to tests, papers, briefings, and most profoundly, this thesis. Her encouragement, logistical support, and love have made this effort productive, and without her, it simply could not have been done.

Tom S. Wailes

Accession For

NTIS GRA&I [✓]
DTIC TAB ☐
Unannounced ☐
Justification PER CALL JC

By
Distribution/

Availability Codes

Dist	Avail and/or Special
A-1	

Contents

Contents

List of Figures

List of Tables

Abstract

A Pascal based ACM proposed standard (CORE) graphics system from the University of Pennsylvania was converted to run on an 8086/8087 microcomputer. Additionally, a non-restrictive full 3-D hidden surface removal algorithm for surfaces modelled by opaque polygons was implemented. This algorithm is an extension of the list priority algorithm created by Newell, Newell, and Sancha. By saving the priority list created by the algorithm from one batch of updates to the next, computational savings are possible. Although the present algorithm can only be used on raster type display surfaces, this algorithm could be used as a prelude to other scan line hidden surface removal algorithms to provide support for vector type displays.

PLACING HIDDEN SURFACE REMOVAL WITHIN THE CORE GRAPHICS STANDARD: AN EXAMPLE

I Introduction

Aircrew survival in today's electronic battlefields is becoming increasingly more difficult. The rise of more sophisticated electronic sensing devices is forcing pilots to fly at lower altitudes and faster airspeeds to reduce their probability of detection. To make matters worse, future battle plans call for aircrews to carry out their sorties independent of weather and lighting conditions, requiring greater concentration by our pilots to avoid natural and manmade terrain features[14:1].

In addition to the strain imposed while penetrating deep into enemy territory, once in the target area, pilots must quickly select from an array of weapon system delivery options and accurately deliver ordinance in spite of new, more accurate target area defenses. As can readily be seen, our pilots are being tasked with more and more decisions with smaller and smaller room for error and need new tools and techniques to penetrate the sophisticated air defenses of the future.

To combat the increasing pilot workload, the Air Force is funding research into the development of more efficient human-aircraft interaction[17:1]. Because of the increasing capabilities of digital electronics, non-traditional approaches to the problem of weapon system operation are being explored. Research areas include voice controlled weapons subsystems, incorporation of human engineering concepts in cockpit layout, and the use of computer graphics to generate information rich, visual displays.

Fortunately, recent developments in computer graphics have aided the development of cockpit graphic displays proposed by the Air Force Flight Dynamics Laboratory. A proposed graphics standard created by the Graphics Standards Planning Committee of the Association for Computing Machinery(ACM) in 1979[16] has motivated vendors and Universities alike to create graphics software that is portable from one installation to another. This standard (commonly called the CORE) defines a set of callable subroutine functions that draw lines, display text, and create different views of user defined scenes on graphics display devices.

In a typical application, a user's program calculates data to be displayed(e.g. the position of an aircraft) and then calls the CORE system to draw the object. The CORE subroutines handle all necessary matrix multiplication and low level communication with the user's output and input devices to draw the object. By separating the user's program and the actual hardware device by the CORE subroutines, the average user doesn't need to know the "guts" of the display hardware, and application programs are device independent.

Problem

The Flight Dynamics Laboratory recently acquired a microcomputer based graphics system to design and test computer generated visual displays for future cockpits. In a preliminary literature search, an existing CORE graphics package that met nearly all of their design requirements was found at the University of Pennsylvania. To decrease the amount of time needed for software development, they wanted to rehost this software, adding a hidden surface removal function to complete the package.

There were three major facets to this problem. First, the University of Pennsylvania (UP) software would have to be modified to allow compilation on the microcomputer system purchased by the Flight Dynamics Laboratory. Second, because the Flight Dynamics Laboratory wanted to display realistic complex scenes that required the removal of surfaces obscured by objects closer to the eye, a hidden surface removal algorithm consistent with the CORE standard had to be chosen and incorporated into the UP software. Finally, a test program needed to be defined which exercised the total graphics system and provided the Laboratory a baseline of system capabilities.

Scope

The software modifications caused by the incorporation of the hidden surface removal modules used the methodology and functional specifications of the ACM SIGGRAPH CORE standard as a guide. Because many graphics vendors support this standard, adherence to this set of guidelines will allow easy future hardware and software improvements. Current cockpit visual display specifications require few graphics input functions, therefore the primary emphasis of this project was the development of rich 3-D graphics output functions for drawing complex images on the cockpit monitors.

Many different methods have been found to model three dimensional objects and perform hidden surface removal[2,3,4,5,6,7,8,9,10,12,13,15,20,21]. Although techniques and algorithms abound, each has limitations and costly drawbacks when used in general applications. Nevertheless, this projects's goal was to implement a modeling system and hidden surface removal algorithm both general in nature and time efficient in most applications. Because of limited time available, the research effort concentrated on developing only polygon hidden surface removal modules for raster type devices. A follow on project is required to develop the modules needed to perform hidden line removal needed by vector type display devices and to perform hidden surface calculations on the other output primitives specified by the CORE.

A simplified test program was written that created images similar to the situation display format proposed by the Flight Dynamics Laboratory. Included in this display were such objects as terrain features, enemy threats, and aircraft position. The test program did not attempt to model aircraft performance or exactly model a portion of the earth's surface. The purpose of the test program was to evaluate the function of the graphics software, not create a detailed data base management system or aeronautical model.

Because the Flight Dynamics Laboratory had recently acquired the hardware for this project, the effort was constrained to use the processing power, system software, data storage, and display capability already purchased.

Overview

This thesis then is a report on a design project that took an existing CORE graphics software package, added a hidden surface removal capability, and then created an application program for test purposes. The following chapter first defines the problem in detail creating a list of design specifications for the final program. The next chapter outlines the alternative approaches possible to solve the problem, including a detailed discussion on the selection of the hidden surface algorithm used in this project. Chapter four contains a detailed technical program description of the hidden surface removal algorithm developed during this research including the mathematical details of the associated comparison tests. Chapter five records performance analysis of the algorithm and the graphics system. Finally, chapter six presents the conclusions reached during the research and and recommendations for future work.

II Detailed Analysis of the Problem and Requirements

As stated earlier, this project had three major obstacles to be overcome: the rehosting of the University of Pennsylvania software, the development and integration of a hidden surface removal algorithm, and the creation and analysis of an application program to test the graphics software. This chapter will take each of these subjects in turn, and reveal the many facets of the problem that are not readily observable.

First, this chapter outlines the problems that were expected when trying to rehost the software; detailing the graphics microprocessor system resources, Pascal language differences, and operating system incompatibilities between the software sent and the graphics system purchased by the Laboratory. Next, the problem of selecting an appropriate hidden surface removal algorithm is discussed, concentrating on the requirements of the CORE standard. Finally, this chapter presents the requirements of a suitable test application program to test the total software package.

Rehosting the U of P Software

The software acquired by the Flight Dynamics

Laboratory was originally developed by the University of Pennsylvania on the Moore School's UNIVAC 1100/61[18:179]. From the beginning, the six member programming team designed the system in Pascal using modular programming techniques to isolate individual functions of the software package. Because they had a modular compilation capability, they additionally decided to place the individual programming modules into separate files. Later when a VAX 11/780 became available, they kept the separate file structure and transferred the system to the VAX computer converting it to VAX Pascal. Therefore, the software source tape that eventually was sent to the Flight Dynamics Laboratory contained over 300 files, all in VAX Virtual Memory operating System (VMS) format.

The microcomputer system obtained by the Flight Dynamics Laboratory for the graphics function consisted of an Intel 8086/8087 microcomputer, 384K bytes of main memory, 512K bytes of pseudo disk(a RAM type memory that appears to the operating system to be a disk), and one double density double sided floppy disk drive having 1.2M bytes of storage. These resources plus a terminal and the SCION display processor (a Z80 based raster scan graphics coprocessor) were available for use in the final operational configuration. All of these resources, were tied together by the CPM operating system which occuppied approximately 15K bytes of main memory.

For software development, the laboratory purchased the

Pascal MT+86 compiler with it's optional screen editor package designed to aid Pascal program development. Development of software occurred on a separate processor from the graphics system that had 512K bytes of main memory, 2.4M bytes of of double density floppy disk storage , and a 10M byte hard disk drive. To allow more than one person to develop software at the same time, these resources were linked together by the MPM operating system (a CPM compatible operating system that allowed multiple users). Software created, edited, compiled, and linked on the large system was transferred to the graphics processor using floppy disks.

Raw Transport of Source Code

Considering the above discussion, the following were the requirements to be met when intergrating the Pascal source code into the Flight Dynamics Laboratory Microcomputers:

1) The tape sent by the University of Pennsylvania was in VAX 11/780 VMS format. The source programs had to be transferred to the Flight Dynamics Laboratory microcomputers. The final code could reside on either the hard disk or floppy disks, however in either case the compiler needed the programs in MPM formated disk files.

2) The transfer of code had to be reliable to prevent wasting precious time debugging code transfer errors.

3) The transfer of code to the micros had to be repeatable should hardware malfunction or operator error inadvertently destroy the transferred files.

4) Because the source tape contained object files and operating system files not needed by the micros, these files had to be eliminated from the transfer process, to save disk space on the smaller capacity microcomputer system disk drives.

Language Incompatibilities

Because the VAX 11/780 Pascal language differed from the MT+86 version of Pascal, a method for changing the source received from the University of Pennsylvania to the language definitions required by MT+86 had to be developed. The following is a complete listing of all the changes needed to perform this activity:

1) The VAX version Pascal defined its program modules using the key word "module" to indicate the beginning of a module and "end." to signal the end of a module. The MT+86 compiler also expected "module" to signal the start of a module, however the sequence "modend." ended a module. The

VAX Pascal compiler allowed module names and procedure names to be the same, therefore, the originators had purposely defined modules with the name of the procedure contained within. The MT+86 linker required that module names and procedure names had to be different. In addition, the VAX needed the Pascal "(input,output)" parameter declarations right after the module name if any input or output was performed by the module. The MT+86 compiler did not require this declaration. Obviously, the "(input,output)" declarations had to be removed from module files that had them, module names had to be replaced with unique names, and the "end." found at the end of modules had to be replaced by "modend."

2) The VAX language used the syntax "% include '" to include files within the modules. The MT+86 compiler expected "{$I }" as its include-file declarations. All include file declarations had to therefore be changed to the MT+86 format.

3) The VAX Pascal placed the keyword "extern" after external procedure declarations for use by the compiler to establish linkages. The MT+86 compiler required the keyword "external" <u>before</u> external procedure declarations. Therefore, all external procedure declarations had to be prefaced by "external" and the VAX key word "extern" deleted. Additionally, the VAX used the key word

"[external]" to preface external variable names where as the MT+86 compiler simply expected "external". Obviously the only change here was to remove the braces.

4) The University of Pennsylvania source writers decided to use the VAX unique "varying" type declaration for the passing of variable length character arrays. This type declaration was equivalent to a packed array of characters, except that string length information was kept by the system in a user accessible area referenced by "userarrayname.length". Unfortunately, they also had to modify the actual program code to integrate strings, especially when initializing strings and comparing two strings.

Pascal MT+86 supported a "string" type much like the "varying" type of the VAX. However, the MT+86 language required the calling of special procedures to initialize ("fillchar") or find the length ("length") of strings instead of the VAX implementation which allowed the user direct access to the array length variable. All procedures that used varying in a type declaration obviously had to be modified by hand.

5) The VAX "case" construct used "otherwise" as the default label to be executed if no matches were found. The MT+86 compiler used "else" as the default. Therefore, all keyword "otherwise" occurrences in the code had to be

changed to "else".

6) The VAX version of Pascal required that global procedure entry points (e.g. the procedure header line such as "[global] procedure mine(...") within modules had to be preceded by the keyword "[global]". The MT+86 compiler did not require this information and therefore these occurrences of "[global]" had to be deleted.

7) The VAX allowed more characters of significance in variable,procedure,and file names than those of Pascal MT+86 and the CPM operating system. This required finding all occurrences of variables and procedures greater than 7 characters (the limit for external variable and procedure names in MT+86) and determining if the new truncated name was unique. All non unique names (e.g. SETLOCP2,SETLOCP3) had to be changed (i.e. SETLCP2,SETLCP3). In addition, filenames greater than 8 characters had to be changed to be compatible with CPM.

Creating New Libraries

To reduce the size of applications programs, the user should only have to link those functions that the program actually needs. Therefore, the software modules created had to be placed into a library that could selectively link the modules. This library had to meet the following requirements:

1) The library scanning function of the Pascal MT+86 linker was a one pass scanning function. Modules had to be placed in the library so that modules requiring the loading of subordinate modules occurred first.

2) There was a large number of modules expected to be in this library. Provision had to be made to either split the modules into separate libraries or provide some other method of access to the assembled modules should the number and size of modules exceed the amount allowed by the linker program.

Developing New Device Drivers

Because the software package developed at the University of Pennsylvania did not use a SCION display processor, new device communication routines had to be created. They had to meet the following specifications:

1) Routines created had to support all output primitives presently supported by the University of Pennsylvania software.

2) Routines created would not require changes to the device independent portion of the University CORE package.

3) Program stubs would exist for all input primitives not implemented on the SCION processor.

Selecting a Hidden Surface Algorithm

Many hidden surface algorithms exist which exhibit varying degrees of efficiency. Most have restrictions which limit their application which is in direct conflict with CORE guidelines. Therefore, the algorithm selected for use would have as a goal the following specifications:

1) Ideally the hidden surface removal subroutines would strictly adhere to the specifications contained in the ACM SIGGRAPH CORE. This implied that the algorithm would only require object descriptions in 3 dimensional world coordinates (user coordinates), the viewing transformations specified by the user, and the viewport specifications given in 3 dimensional Normalized Device Coordinate space(NDC)[16:III-9]. As suggested by the CORE, it would be implemented at the interface between NDC and the physical device[16:III-11]. Finally, it would support the four hidden surface implementation levels defined by the CORE[16:III-12]:

a) None: no hidden surface processing.

b) Temporal: Objects drawn strictly in the order given by the user.

c) Explicit: Objects displayed prioritized according to which z plane they have been assigned to by the user. This is commonly referred to as 2 1/2 D.

d) Full : Full hidden surface removal for all output primitives.

2) Because the CORE central concept is generality of use, the algorithm selected would allow the picture generation of almost any image. This implied that it be free from restrictions commonly found in hidden surface algorithms[7:10]:

a) It would allow polygons of an arbitrary number of points(some algorithms require that polygons only have 3 sides).

b) It would allow penetrating polygons.

c) It would allow concave polygons to be modeled.

d) The user would not have to specify whether the polygon points are defined in a clockwise or counterclockwise direction.

e) The user could define polygons that do not belong to polyhedrons.

f) There would be no restriction to the number of polygons that share the same edge (e.g. a paddlewheel of planes is permitted).

g) Boundaries with internal faces would be allowed.

h) Line segments would be allowed.

i) The definition of objects would not require more information than a list of vertices.

3) Because the algorithm would most probably be used by programmers that knew little about the complexity of the hidden surface removal problem, the algorithm had to be robust, allowing inexperienced programmers to get results even if they made minor errors. This implied that the algorithm accept non-coplanar points and additionally be free from infinite loops on complex objects that had inadvertent intersecting planes.

4) Additionally, the algorithm had to be computationally and memory space "efficient" (e.g. able to run on non virtual memory systems) in a wide range of applications. This requirement ensured that even users of small systems would be able to get high quality graphics pictures created by the CORE.

5) Finally, the algorithm selected would allow incorporation of shading and other visual realism techniques "on top of" the CORE (provided by vertex color intensity arrays), to enhance the usability of the CORE in high cost, computationally rich systems that could afford these capabilities.

Application Program Design

Because this project involved the rehosting of a major software package and the incorporation of a major addition to the package, a test program would be needed to exercise the package. Specifically, the test program would create an image that satisfied the following specifications:

1) Display the situation display format:

a) Simulated terrain data, consisting of four sided polygon planes with the points corresponding to the terrain elevations at 1/4 mile increments, would be displayed. Other possibilities include: increasing the resolution of the display by making the terrain squares smaller, defining the terrain by shapes that had the same altitude (e.g. a relief map), and using some type of polynomial to compose polygons that approximated the surface. The 1/4 mile four sided polygons were chosen simply for convenience.

b) Several threat markers would also be displayed consisting of red and yellow polyhedrons corresponding to enemy anti-aircraft positions. These polyhedrons would contain at least 8 polygons and would be placed in a separate data base from the terrain data.

c) An aircraft marker made of a triangular body and a small triangular tail surface would indicate aircraft position.

d) The eyepoint for the display would be set 7,000 ft behind and 1,000 ft above the aircraft position.

2) The view provided by the system would be a perspective view with hidden surfaces removed.

3) The system would generate the illusion of motion by moving the aircraft and eyepoint through the terrain database, drawing new views after each move.

This chapter has defined the requirements that the graphics system must meet. The next chapter discloses the approach taken to meet them.

III Solution Decisions

Deciding on the proper course of action to solve the problems outlined in the previous chapter was not easy, due to the many possible approaches at various levels. However, this chapter attempts to justify the decisions made to create the final system. This chapter will first describe why a Pascal program was written to automate the many changes needed for the conversion of the University of Pennsylvania source code into a form compilable on the Laboratory's microcomputers. Next, the reasoning behind choosing a depth sorter type hidden surface removal algorithm for this package is discussed. Finally, this chapter comments on the development of a suitable testing sequence to debug the final system.

Rehosting the University of Pennsylvania Software

Because this research needed a reliable, repeatable and self documented conversion of the software package originally received from the University of Pennsylvania, rehosting the software was accomplished in a two step process. First, the files contained on the VAX VMS tape were transferred onto floppy discs and then a conversion program that took the floppy disc files and converted them into a compilable form. In the end, this process required the use of two VAX 11/780's in addition to the Laboratory's microcomputer system.

The first problem was of course to find a computer which could place the source tape onto floppy discs in a CPM/MPM format. The main AFIT UNIX VAX 11/780 was known for its ability to create CPM formated floppy discs and was initially chosen to make the conversion. Unfortunately, the tape reading facilities of its UNIX operating system did not understand VAX VMS format and when attempts were made to read the original tape, the resulting files contained many extraneous characters.

To clear this hurtle, I found another VAX 11/780 located at the Air Force Avionics Laboratory that had a VMS operating system and a special utility program that created formated tapes readable by the UNIX operating system. By using this program, I created a tape readable by the UNIX VAX and then used the UNIX VAX to create the needed floppy discs.

Because the MT+86 Pascal compiler of the Laboratory had string searching, string deletion, and string insertion system subroutines available to the user, the decision was made to automate the changes needed for the Pascal language differences by writing a conversion program. This method of conversion had several advantages over other methods including:

1) It would be easily repeatable if for some reason

the converted files were destroyed, or if the University of Pennsylvania sent a more recent release of the system.

2) It would automatically document all the changes needed during the conversion process.

3) It would catch all occurrences of change. Manual editing would certainly miss some needed changes.

It is debatable whether or not this method took less time than a simple editing session method to make the initial changes. However, during the compilation process when changes were found to be necessary that were not initially known, the conversion program was simply changed slightly and run again. This time spent modifying the conversion program was small compared to the time that would have been spent to search and edit the entire module library for later required changes. In addition, this method taught the author how to use the system subroutine library, the Pascal MT+86 compiler, and the system subroutine linker program which sped later software development.

Developing libraries to separate the functions for size consideration proved to be more difficult than originally expected. It was found that the MT+86 system allowed only 64K bytes of program area for any program. An initial compilation revealed that 32K bytes of program area was needed for a basic repetoire of CORE subroutines to set color, create and delete retained segments, cause newframe actions, batch updates, draw polygons, lines, and text, and allow setting a fairly complete viewing specification. Additionally, 21K bytes of device driver software was needed to support the above mentioned device independent CORE routines. Finally, Pascal system routines took up an additional 14K bytes of memory to support the package, resulting in over 67K bytes of program area without hidden surface routines or a user's application program. This was clearly a major problem and some solution had to be found.

To solve this problem, all graphics input functions were removed and the CORE graphics and device driver subroutines were placed into three independent overlays (program groupings that are read from a secondary memory device only when needed). This meant analyzing each of the more than 250 program modules and determining where in the CORE input and output pipeline they where needed. After making the decision as to the overlay number of each module, the overlay number was placed into its Pascal MT+86 external procedure declaration and the file name was placed into a special overlay library build file. Three overlay areas were defined: Initialization/ Termination, Device Independent, and Device Dependent/ Hidden Surface.

Overlay one was rather small and contained the procedures used one time only for initialization and termination. This freed the two main overlays from the burden of infrequently called procedures and gave them more room for development and debugging. Overlay two contained most of the user callable CORE procedures that set viewing parameters, drew lines, drew polygons etc. along with the utility procedures needed to support them such as the graphics "clipping" routine. Overlay three contained the device dependent device driver routines along with the hidden surface routines developed during this research. Finally, there were a few routines that were needed by more than one overlay group such as the matrix multiplication routine. These few routines were placed into the "root" or main program by a user hidden, never called subroutine that was incorporated into the global variable include file to create the compiler references for them.

This definition of overlays needed only two disk accesses per picture to draw scenes. A sample scenario of operation would start by accessing the initialization overlay, caused by a user call to initialize the CORE. After the initialization was complete, the first call to a device independent routine would bring in the second overlay. This overlay would remain in memory while the user defined more polygons, lines, text etc. Upon calling END_BATCH_OF_UPDATES, the third overlay would be read into memory and the actual drawing of primitives on the device would be accomplished for this batch. Thus per frame there would only have to be one swap of overlays.

All communication between the overlays was accomplished using data structures "owned" by the root. By defining these data structures in the root as local and by every other module as external, simple coordination of effort was possible. Finally, when the user finished a session, a call to TERMINATE_CORE would again read in overlay one, this time to perform the proper termination actions on the display device.

Selection of a Hidden Surface Removal Algorithm

Because this research's goal was a general purpose hidden surface algorithm that was modular in design, memory space efficient, free from the usual hidden surface removal algorithm restrictions, and computationally efficient, the Newell, Newell, and Sancha algorithm[13] was selected for study. This algorithm simply sorts all polygons according to their maximum z coordinate, resolves any ambiguities when the polygons' z extents overlap, and then scan converts each polygon in descending order of largest z coordinate. By "painting" the objects furthest from the eye first and then overlaying them with polygons nearer the eye, a technically correct image is produced taking advantage of the overwriting characteristics of raster devices.

A review of hidden surface removal algorithms published in the literature revealed that because of the computational cost of generality, most algorithms had some type of restriction. In a significant article written in 1974 the authors Sutherland,Sproull, and Schumacker compared 10 well known hidden surface algorithms including the Newell, Newell, and Sancha algorithm[20]. In this study, it was revealed that only the Warnock, Watkins, Bouknight, and Newell,Newell, and Sancha algorithms permitted the user to draw images in an unrestricted environment[20:36-7]. The other algorithms proposed by Appel, Galimberti, Loutrel, Roberts, Schumacker, and Romney all had some combination of restrictions, most commonly being the requirement for additional information (adjacent edges, clustering of faces, etc.) and the disallowing of penetrating polygons. Because of my goal for generality of use, these last six algorithms were removed from further consideration.

Being able to draw arbitrary images is certainly an honorable goal, but naturally no one wants to wait for minutes for a slow hidden surface algorithm to draw a cube. In fact, it has been the quest for a simple computationally efficient algorithm that has driven researchers to explore many differing approaches to solving the hidden surface removal problem. Sutherland, Sproull, and Schumacker addressed this important issue of computational efficiency, and in their paper they proposed a computational cost ordering of the ten algorithms previously mentioned in three hypothetical environments [20:54].

The first imaginary environment, a characterization of most algorithm testing scenes, consisted of a simple house made of some 200 polygon faces. An intermediate test environment called "a harbor scene" consisted of some 5000 polygon faces and was intended to represent what one would expect on a ship simulator scene of New York harbor. The final environment proposed was the same harbor scene with small detail added such as windows in buildings, finer curved object approximations, and was intended to contain a staggering 120,000 polygon faces.

Although the study was simply an order of magnitude calculation of how each algorithm would perform, the Newell, Newell, and Sancha algorithm was considered an order of magnitude more computationally efficient than the ten other algorithms in the first environment, twice as fast as its nearest competitor in the second environment, and about 5 times slower than the Romney algorithm (deemed the winner) in the third environment[20:54]. However, Romney's algorithm assumed all polygons were convex, required all polygons be expressed as triangles, did not allow penetrating polygons, and therefore was removed earlier from my list of possible algorithms. Of the general algorithms still remaining under consideration, the Warnock algorithm was fastest in the third environment with the Newell, Newell, and Sancha algorithm taking about 1.65 times as long to compute the picture. Because its performance would be theoretically superior in applications up to 5000 polygons and in the extreme still remain within an order of magnitude of other general hidden surface removal algorithms, the Newell, Newell, and Sancha algorithm was selected out of this initial study as my candidate for research.

Obviously the search for an efficient hidden surface algorithm did not stop in 1974, and in fact, there have been many new algorithms proposed since that time. However, in my search of the literature, I've only come across one documented report of a hidden surface removal capability residing in a CORE standard system[11]. This particular package used an algorithm for hidden surface removal proposed by Myers[12]; a raster-scan class algorithm that can vary the number of pixels used in generating a picture[11:81]. Unfortunately, this algorithm required a large amount of memory for even a rather modest hundred polygons[11:81-2] and therefore this method was removed from my list of alternatives.

The majority of algorithms developed since 1974 tend to be geared toward one particular application of computer graphics or toward one particular set of problems. This can clearly be seen in the area of animation where several algorithms try to decrease the time spent generating successive picture frames by investing a substantial amount of time in preprocessing the data base to be displayed. These methods, similar to the concept proposed by Schumacker in 1969, depend upon a priori knowledge of the scene and may require direct operator manipulation of the database to minimize scene creation time[15:113]. Another restriction found in these types of hidden surface speed up techniques was the limitation of the scene to only static objects or motion only under certain conditions, with the animation coming from movement of the eye point through the data base[9][5]. This limitation may indeed be acceptable for flight simulators but some CORE applications may require movement of objects within the scene.

Special techniques exist that perform hidden surface elimination on surfaces defined by equations[2](which the CORE does not support[16:III-75]), on voxel based representations of medical data[1], and on special chemistry molecular models[10]. Other algorithms exist which claim computational efficiency but require triangular polygon definition[3], require polygon edges to belong to at most two polygons[8], and strictly prohibit penetrating polygons[6:264]. All of these non-generalized techniques were discarded in favor of the Newell, Newell, and Sancha approach. It is interesting to note that although some of these methods were discarded because of their restrictions, at least two algorithms devoted to fast computation of animated scenes used the general list priority creation, paint from back to front, split polygons if necessary, steps of the Newell, Newell, and Sancha algorithm[4][5].

As with all good things, this algorithm was known to have some drawbacks and indeed would not solve all problems. First of all, this algorithm is not well suited for raster scan film recorders. Once the film is exposed on these output devices, the film cannot be overwritten with the new values for polygons that obscure those previously drawn. Also this method is effective only for hidden surface removal and would not produce hidden line drawings on calligraphic displays. However, this algorithm could be used as a prelude to the excellent scan line algorithm developed by Watkins[21].

As suggested by [20:42], the Newell, Newell, and Sancha algorithm could be used to compute a priority ordering of all polygons in the scene. Instead of actually scan converting each polygon as it was resolved, the list of polygons could be saved in a linked list maintaining the order of "painting" required to generate a correct image. After all polygons had been ordered, an index number could be placed into the data structure by sequentially numbering the polygons from back to front. The new data structure could then be ordered into the y bucket sort needed by Watkins, and his algorithm could begin scan converting each line. During each scan line span calculation, when it was necessary to determine which polygon was in front of another, a simple reference to the index number would resolve the question. This simple test would replace the more complicated tests used by the Watkins algorithm.

It is difficult to predict the performance of such a hybrid type algorithm, however, it would allow the correct rendering of hidden line pictures and would create the type of output needed by raster type film recorders. Unfortunately, one disadvantage caused by such an algorithm would be the loss of parallelism inherent in the Watkins algorithm (e.g. resolving polygons while scan converting them). If however, the user is generating a sequence of images, it would be possible to create a hidden surface pipeline having two processors. The first processor would perform the Newell, Newell, and Sancha algorithm, ordering primitives in a "resolved primitives" file. The second processor would take this file and scan convert each primitive using the Watkins algorithm. This division of labor would allow the simultaneous resolving of the display priority of the next batch of updates while scan converting the present batch of updates.

Testing the Graphics Package

The testing of the graphics package was divided into three phases. First, an initial interactive program was developed that drew lines, polygons, and placed text on the screen. This program was used to test and debug the device driver routines needed by the package. This program also served to validate the basic features of the CORE package, insuring that the system had remained intact during the Pascal conversion process. It was during this phase that it was found that a totally different color selection scheme would be required to draw color pictures.

The University of Pennsylvania's display device achieved color by drawing pixels with separate calls to draw the red, green, and blue values of the primitive (e.g. a line). The SCION, on the other hand, used a color lookup table which had to be predefined by the user and needed only a selection of the color wished to be drawn. This seemingly small change caused changes in some 21 different modules.

In addition, an application program to debug the hidden surface routines and determine the cost of performing hidden surface removal had to be developed. To initially test the hidden surface removal package, a simple scene of intersecting cubes was chosen for display. By coloring the sides of the cubes different colors, it was obvious when the algorithm performed properly. Several object creating routines were developed to perform this test, so that simple increments in scene complexity could easily be accomplished to test the routines under varying scene conditions. Additionally, this method would easily allow testing the package with several batches of updates to determine if image transformations and retained segment merging of primitives occurred as expected.

Finally, after this initial testing was completed, a program to draw the situation display format was created to analyze the systems usefulness to the Laboratory. Because the Laboratory had neither the computational resources of a mainframe computer nor special purpose graphics support hardware, the decision was made to initially limit the number of polygons displayed to around 250 (comparable to a preliminary study done by Boeing for the laboratory). This would still allow over 200 terrain faces and would allow testing of the algorithm with several objects used in their scenario.

Now that the reasons behind my major design decisions have been discussed, its time to actually reveal the details of my hidden surface removal algorithm.

IV Detailed Algorithm Description

The previous chapters of this work describe the initial steps common to any major software effort, namely the requirements definition and preliminary design phases. These steps must be accomplished before the final detail work begins to prevent wasted effort chasing infeasible solutions and creating unneeded system frills. Using the past chapters as a foundation, this chapter describes the actual working program.

First, a short background section describes the concepts used by the CORE specifications to define objects to be displayed and points out additional burdens placed on hidden surface removal algorithms by the CORE. Additionally, this section briefly describes the data structure and output viewing pipeline used by the original package to give the reader an understanding of the data available to the hidden surface removal algorithm.

Second, the programming steps required by my algorithm to create a correct rendering of the CORE specified scene are outlined. As stated earlier, this algorithm is an extension of the algorithm first proposed by Newell, Newell, and Sancha which basically "paints" objects from back to front, letting the natural overwriting properties of raster devices resolve conflicts. In addition, an improvement to the algorithm proposed by Sutherland, Sproull, and Schumacker is discussed.

Object Description Within the CORE

Object descriptions within the CORE can be considered on two different levels. To allow easy interactive work (e.g. moving objects around on the screen) the CORE system requires the user to specify a beginning and end to object definition. For example, if the user wishes to draw a house made of several lines he must first open a "segment". He then issues commands that draw the lines comprising the house, and then he "closes" the segment. This outer layer is useful if, for example, after the house is drawn the user wants to remove the house without having to redraw an entire village.

Within each segment, there is another level of description called the "primitive" level within which the user actually issues the commands to draw objects. Six basic output primitives are used to actually produce images on the display surface, namely: line,polyline,text,marker, polymarker, and polygon (move is also a primitive, but it does not affect the screen). Provision must be made by the hidden surface routines to perform hidden surface removal on all six of these primitives.

Most hidden surface removal algorithms described in the literature only discuss the problem of correctly rendering correct scenes made of polygons and make no mention of provisions for line, text and marker primitives required by the CORE. Although not presently implemented, the Newell, Newell, and Sancha algorithm can be extended to handle these special cases with little change in the algorithm flow.

Because we are writing to a raster device, all we have to do is model the marker and polymarker primitives as points in space. Using the depth sort process we will simply draw these objects at their proper depth priority and let objects closer to the eye overwrite them if needed. Lines will simply require additional checks to determine if intersections occur with polygons. If an intersection does occur, the line will be divided at the point of intersection and the two line halves and the polygon will be inserted in the hidden surface linked list and output primitive data structures. Polylines will be considered as collections of individual line segments for this processing. Finally, text string primitives will simply be treated as a 3 dimensional rectangular polygon that is defined by the extent box surrounding the string of individual characters. Calculating the polygon vertices will require the interrogation of the present values of the CORE "CHARUP,CHARPATH, CHARSIZE, CHARSPACE, AND CHARJUST" attributes [16:II-22 - II-28].

Describing the data structure provided by the University of Pennsylvania CORE system requires us to remember our previous description of segments and output primitives. In their data structure, a linked list is kept of all the segments created. Within each segment's record, information that affects all the primitives in the segment is kept, including: its image transformation, its name, and detectability. Each segment points to a linked list of associated output primitives. These output primitive records contain the coordinates and other related attributes of the primitive needed to draw the line, polygon, marker,etc. on the viewing surface. Thus we have a linked list of output primitives that make up a segment, and a linked list of segments that make up a picture.

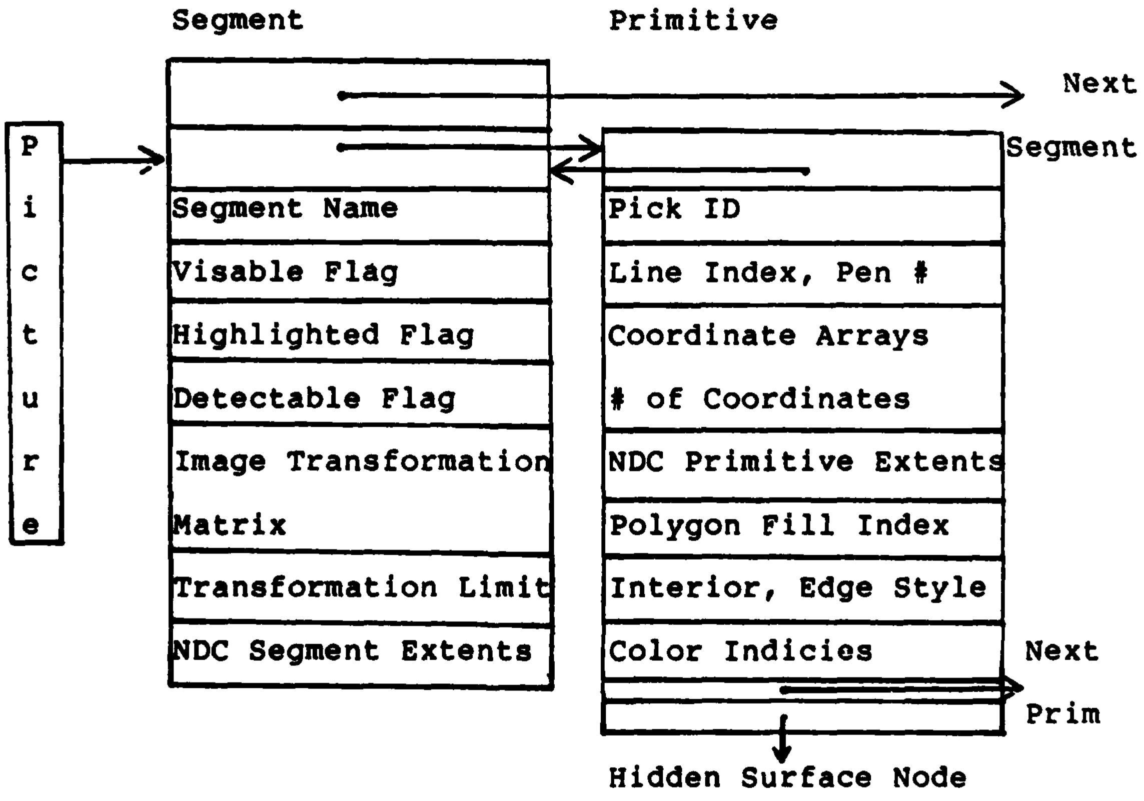


Fig 1. Segment and Output Primitive Data Structure

To invoke hidden surface removal, the user must tell the system which objects should be processed. This is accomplished using the Begin_Batch_of_Updates and End_Batch_of_Updates CORE subroutines. Segments defined after the batch begins will not be drawn until End_Batch_of_Updates is called. At this time, previously defined retained segments and all new segments defined in the batch of updates will be depth sorted and processed to create a new scene.

When each new primitive is added to the linked list within each segment, the user defined points are transformed into CORE system Normalized Device Coordinates(NDC) and left in this coordinate space for future use by the final image transformation, hidden surface, and display routines. This transformation is a concatenation of four operations consisting of a multiplication of the image by a user defined instance transformation (a user rotate, scale, translate), multiplication of the image by another 4X4 matrix to place the image in the viewing plane, "clipping" the points to a user defined viewing volume, and then mapping the clipped points into a user defined "viewport" on the viewing surface. This entire process is called the "viewing operation" and details of this process can be found in [3:II-48 - II-70].

To speed computations requiring the generation of a series of scenes in which a segment moves in relation to its original orientation, the CORE standard allows the user to specify an image transformation after the viewing operation and before the final mapping to screen coordinates. During each batch of updates, the user defines viewing specifications and issues output primitive commands which invoke the CORE viewing operations of normalization, clipping, and mapping to the user's viewport. The NDC coordinate results of this operation are then saved in the main segment and primitive data structures which retain each segment's coordinates until they are removed by a DELETE_RETAINED_Segment action. The CORE final image transformation takes these NDC coordinates and rotates, scales, and translates each segment's image using a 4X4 matrix whose values are computed from rotation, scaling, and translation parameters passed by the user. By simply varying the image transformations from batch to batch the user can cause segments to move (e.g. the flaps on an aircraft) or can simulate eyepoint motion (true motion requires a non-linear transformation that this method really could not provide). Because the user only has to change the segment's image transformation and does not have to perform the entire viewing operation, this method is significantly faster than having to redraw all images during each batch.

The present implementation performs all of its hidden surface calculations on the image transformed coordinates of each primitive. This allows the user to take advantage of the processing speed-up due to image transformations and still receive true renderings of each scene. Additionally, because many batches of updates may be needed by the user to create an animated sequence of pictures, my extension of the Newell, Newell, and Sancha algorithm maintains the priority list and polygon definitions from the previous batch of updates in the hope that the priority list will change very little from batch to batch and that few polygon subdivisions will be necessary if the object orientations don't radically change. This idea to improve the algorithm was first introduced by Sutherland, Sproull, and Schumacker in [20:40].

The Algorithm

The purpose of the algorithm developed during this research is to create an ordered list of polygons that when scan converted on a raster device, will result in a proper image. To accomplish this task, three basic steps are needed. In the preprocessing step, an initial ordered list ordered by maximum z is created. This list is further refined by the resolving step which resolves any ambiguity in primitive ordering. Finally, as ambiguities are resolved, output primitives are scan converted and placed on the viewing screen.

Preprocessing

During each batch of updates, the user may add new segments and associated primitives to the list of segments that comprise a picture. As each primitive is added, a pointer that will eventually point to a hidden surface node is set to nil. This pointer serves two purposes; first it allows rapid deletion of the hidden surface structure associated with each output primitive when deletion becomes necessary and additionally it serves as a flag to determine if hidden surface initialization has been performed (e.g. a nil pointer indicates that hidden surface initialization has not been performed for this primitive). This is the only additional data item or field needed in the primitive data structure to support hidden surface removal and is the only data area overhead incurred by users who are not calling the hidden surface subroutines.

The actual interface from the CORE to the Hidden Surface support routines occurs within the END_BATCH_OF_UPDATES subroutine and the delete primitive subroutine. When END_BATCH_OF_UPDATES is called and the full hidden surface attribute has been set, a call is made to the SORTER subroutine. This subroutine creates a linked list structure that will be used as input to the RESOLVER or main driver routine of the algorithm.

This interface can be discribed by the following pseudo code:

```
END_BATCH_OF_UPDATES:
     IF FULL HIDDEN SURFACE THEN
        BEGIN
          CALL SORT_HIDDEN    (* PRODUCE ORDERED LIST *)
          CALL RESOLVER        (* RESOLVE AMBIGUITIES  *)
        END;
```

As a first preprocessing step, the SORTER procedure examines the primitives within each segment. If the hidden surface pointer is nil, a call is made to the hidden surface initialization procedure which calculates and stores the x and y screen extents (minimum and maximum), the z extents in NDC space, and the plane equation coefficients for polygons in a separate hidden surface removal data structure (Figure 2). Because the CORE specifications allow the user to perform a final image transformation to the stored NDC coordinates before display on the screen, all of the above calculations must be based on the image transformed points. Additionally, because polygon splitting may be needed later, the initialization procedure must save the image transformed points in the hidden surface data structure.

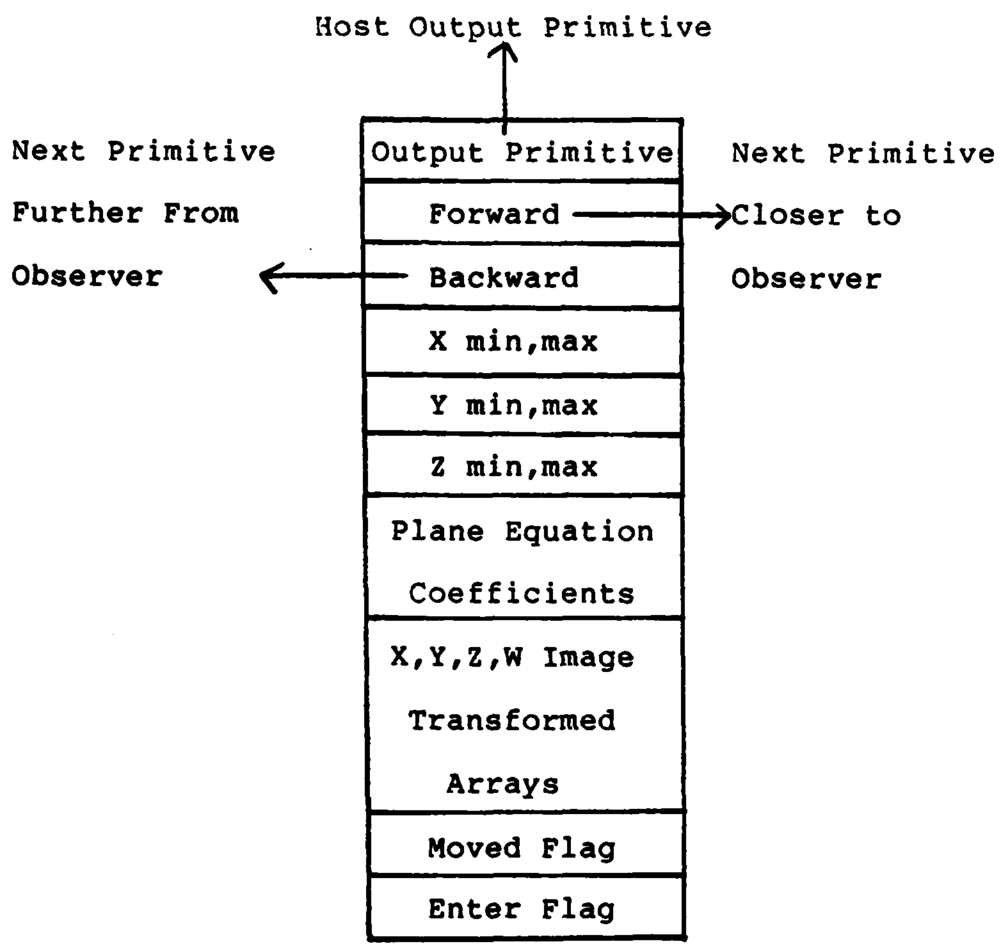

Host Output Primitive
Next Primitive Further From Observer
Output Primitive
Forward
Backward
X min,max
Y min,max
Z min,max
Plane Equation Coefficients
X,Y,Z,W Image Transformed Arrays
Moved Flag
Enter Flag
Next Primitive Closer to Observer

Fig 2. The Hidden Surface Linked List Data Structure

The plane equation coefficients which define the plane of a polygon using the equation $aX + bY + cZ + d = 0$ can be calculated using several techniques. For example, if three points on a plane are known not to be collinear, then the a, b, and c plane equation coefficients can be computed by taking the crossproduct of two edges between them. Because such a computation requires detection of special cases and because an inexperienced user may introduce non-coplanar polygon descriptions, the calculation of the plane equation coefficients used in this implementation is based upon a method developed by Martin Newell[20:14-15]. Using this method, the a, b, and c plane equation coefficients for the list of vertices $V_i = (X_i, Y_i, Z_i)$ can be found by evaluating :

$$j = (\text{if } i=n \text{ then } 1, \text{ else } i + 1)$$

$$a = \sum (Y_i - Y_j)(Z_i + Z_j)$$
$$b = \sum (Z_i - Z_j)(X_i + X_j)$$
$$c = \sum (X_i - X_j)(Y_i + Y_j)$$

In the present implementation, the fourth coefficient d is found by substituting the X, Y, and Z coordinates of the first point in the vertex list along with the a, b, c parameters calculated above into the standard plane equation $aX + bY + cZ + d = 0$ and solving for d. An "average" d could be computed by summing the d for all of the points and then dividing by the number of verticies. However, this would require 3 floating point multiplications and subtractions per vertex and would only help approximate the d coefficient for non-coplanar points. Because non-coplanar points would only be input as an error by the user, the time needed to find an average d was considered unjustified.

However, the present method is quite good and in fact

"...this method requires only one multiplication per coefficient per edge. If the polygon is not planar, this method will produce a plane equation related closely to the polygon, but not the best fit plane equation."[20:15] Because of the ability to get reasonable results even when the user inputs non-coplanar points, this method of calculating the plane equations helps make the algorithm more robust.

Because the CORE specifications allow the user to define polygons in either a clockwise or counterclockwise direction, a simple modification to the plane equation coefficients may have to be made. It can be shown that the plane equation coefficients created for a polygon defined in a clockwise direction will be opposite in sign and equal in magnitude to those coefficients created from the same list of vertices except defined in a counterclockwise direction. In other words, the same locus of points can be defined by plane equation coefficients whose signs for a,b,c, and d are reversed (e.g. $aX + bY + cZ + d = 0 = -aX - bY - cZ - d$).

This causes complications because all polygon plane tests will be performed by substituting the vertices of some polygon A into the plane equation of some polygon B to determine if the vertex of A is closer or farther away from the observer than the plane of polygon B. A convention is needed such that no matter in which direction polygon B is defined, the test results from substituting polygon A's points into polygon B's plane equation will correctly decide which is further from the observer. This convention is fortunately easy to accomplish.

By arbitrary decision, we establish the convention that a point A which is on the same side of the plane as the observer's eyepoint must yield a positive plane value when substituted into the plane equation of B (e.g. (plane_value = $a_B X_A + b_B Y_A + c_B Z_A + d_B > 0$). Since this holds when point A is the eyepoint (i.e. the origin $X_A = Y_A = Z_A = 0$) this convention requires that $d > 0$. Therefore, all we have to do is check the plane coefficient d computed above. If d is negative then the signs of all of the plane equation coefficients must be reversed, otherwise no modifications of the plane equations are necessary.

For example, consider the locus of points parallel to the xy plane located at z = -2 and imagine an observer at the origin (0,0,0). The plane equation for these points is simply a,b,c,d = [0,0,-1,-2] or [0,0,1,2] (for an exercise substitute the points (x,y,z) = (-1,3,-2) and (5,30,-2) into aX + bY + cZ + d = plane_value using both series of plane coefficients and see that plane_value is 0 indicating that both points are on the plane). The point (5,30,-1.5) should be on the same side of the plane as the origin (the observer). If you substitute (5,30,-1.5) into the first set of coefficients you will obtain a plane_value of -0.5 and if you substitute it into the second set you will obtain +0.5. Therefore, if the user had defined an object whose plane equation resulted in the first set of coefficients, reversing the signs will produce the testing values desired.

As shown in figure 2, three pointer values are used by the hidden surface linked list structure to connect the hidden surface nodes to the main CORE segment structure and provide display prioritization of the primitives. The OUTPUT_PRIMITIVE pointer points back to the host primitive of this hidden surface node and allows the primitive data structure to remain the principle repository of output primitive information. The FORWRD and BACKWRD pointers are used to order the primitives with FORWRD pointing to the primitive next closer to the observer and BACKWRD pointing to the primitive next further from the observer as compared to this node. The initialization routine sets the OUTPUT_PRIMITIVE pointer and initially sets the FORWRD and BACKWRD pointers to nil.

The remaining two hidden surface data structure entries MOVED and ENTER are used to mark primitives that have been moved within the sorted list (and thus no longer ordered by maximum z) and to denote how this node entered the list. Nodes created during preprocessing by the initialization routine are labeled NORMAL, those made during a previous batch of updates are labeled UPDATE. Finally, those primitives created during polygon splitting are labeled SPLIT or NEW_PRIM depending on whether they are on the negative or positive side of the split plane at split time (new entries into the list are on the positive side).

Upon return to SORTER from the initialization routine, the newly created hidden surface node will be placed into a temporary linked list of new hidden surface nodes called the "mergelist". Primitives whose hidden surface pointer was not nil (made on previous batches of updates) will be "updated" by performing any image transformation requested by the user and then appropriately calculating new plane equations, extents, and storing the new points in the hidden surface node. Thus, when all segments and primitives have been processed, there will be two linked lists: the updated previous batch of updates' hidden surface linked list, and the new input mergelist of entries.

The SORTER then merges the new hidden surface nodes from "mergelist" into the previous batch of updates' linked list, sorting by the maximum value of z in NDC space. An important point to remember here is that because primitives marked as moved from previous batches are out of order in z, they are ignored during this sorting phase. Of course, this new list will undoubtably contain some ambiguities caused by the merging entries, however, only small changes should have to be made to the list to draw it in the proper order.

The following briefly summerizes the sorting process:

```
SORT_HIDDEN:
     WHILE MORE SEGMENTS DO
      IF SEGMENT VISIBLE THEN
         WHILE MORE PRIMITIVES DO
           IF NO HIDDEN SURFACE NODE THEN
              BEGIN
                INITIALIZE NODE;
                SAVE ENTERING NODE IN MERGELIST
              END
           ELSE
              UPDATE NODE WITH NEW IMAGE TRANSFORMATION
      ELSE
         WHILE MORE PRIMITIVES DO
           IF HIDDEN SURFACE NODE THEN
              DELETE HIDDEN SURFACE NODE FROM HIDDEN
                                   SURFACE LINKED LIST;

      WHILE NODES IN MERGELIST DO
         PLACE NODE IN HIDDEN SURFACE LINKED LIST
                          ORDERED BY MAXIMUM Z VALUE;
```

Resolving the Linked List

After the new merged list of output primitives is created, the individual polygons are examined to determine if polygon reordering or polygon splitting is necessary to paint the picture in the proper order. This examination involves selecting a polygon P and comparing it against the set of all primitives Q that overlap it in z (Figure 3). If P does not obscure any primitive in this set then we can scan convert P, write it on the screen, and allow the natural overwrite process to create the correct image.

The tests used to determine if P obscures the next primitive in the set Q are purposely placed in increasing order of complexity and consist of six basic comparison steps, namely:

1) Is the next polygon in the list a member of the set Q (P and Q_i z extent overlap)?

2) Do the x extents of P and Q_i overlap?

3) Do the y extents of P and Q_i overlap?

4) Is P behind the plane of Q_i?

5) Is Q_i in front of the plane of P?

6) Do the screen projections of P and Q_i overlap?

If it cannot be proved that P does not obscure Q_i then either P or Q_i is split into two primitives and the resulting primitives are placed into the hidden surface linked list ordered by z. Polygon splitting is expensive in both computation time and data storage space, and therefore is avoided if possible.

The first test determines if all ambiguity has been resolved between P and the set Q that overlaps it in z extent. This is done by examining the z maximum of the next polygon on the hidden surface linked list closer to the observer (Figure 3,4a). If P's minimum value is greater than the maximum value of this new test polygon then we have reached the end of the set Q and can scan convert P (as shown by Q_5 in Figure 3).

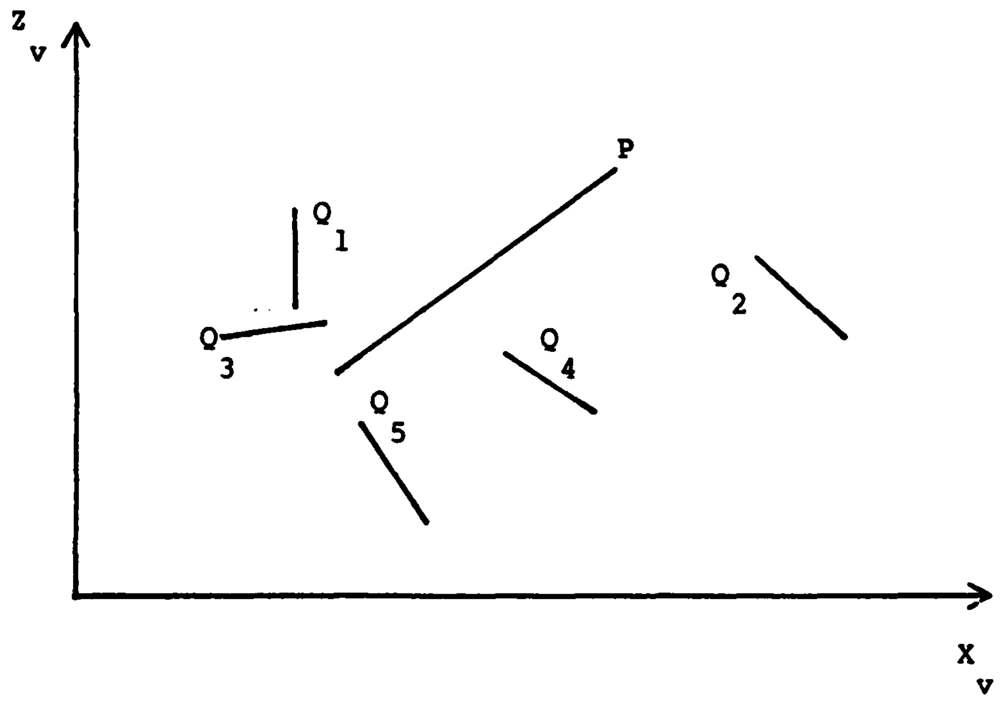


Fig 3. Initial Ordering

If this first test fails, a comparison is made of the x extents of P and Q_i on the screen surface. This is done by comparing the screen x extremes calculated earlier in the initialization procedure. If P is to the left of Q_i (P's maximum x is <= Q_i's minimum x) or if P is to the right (Q_i's maximum x is <= P's minimum x) then Q_i cannot possibly be obscured by P (Figure 4b). Under these conditions the Q_i pointer is advanced to the primitive next closer to the observer and testing begins once again. If the x extent fails, then a similar test is made for the y extents to determine if the two primitives overlap in y (Figure 4c).

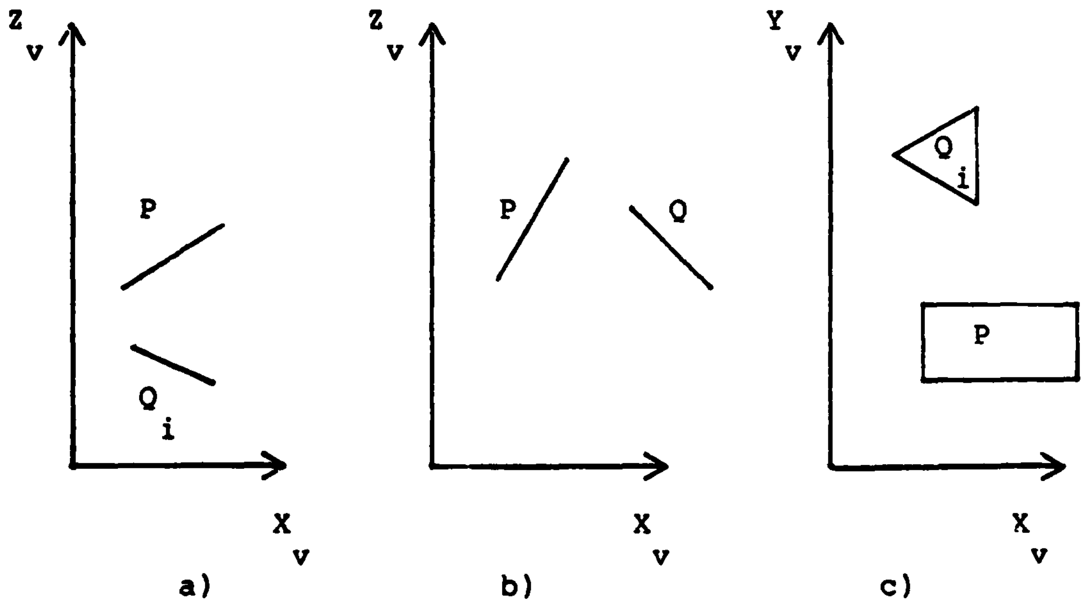


Fig 4. Extent tests

If Q_i has not been eliminated from consideration, the plane equations of the polygons are used to resolve the ambiguity. The first test determines if the polygon P has all of its vertices on the "back side" of Q_i (that is, on the opposite side of the plane of Q_i from the observer). This is performed by evaluating the formula $aX + bY + cZ + d = value$, for each point of polygon P where a,b,c, and d are the coefficients of the plane equation of Q_i and X,Y and Z are the coordinates of the point being evaluated. If all vertices substituted into this equation produce a negative result, then the polygon P lies wholly on the back side of Q_i and should be scan converted before Q_i (Figure 5a). If this test fails, a similar test is performed to determine if the vertices of Q_i are all on the "front side" of P (Figure 5b). The only difference here is that all the Q_i vertices must produce a positive result when placed into the plane equation of P.

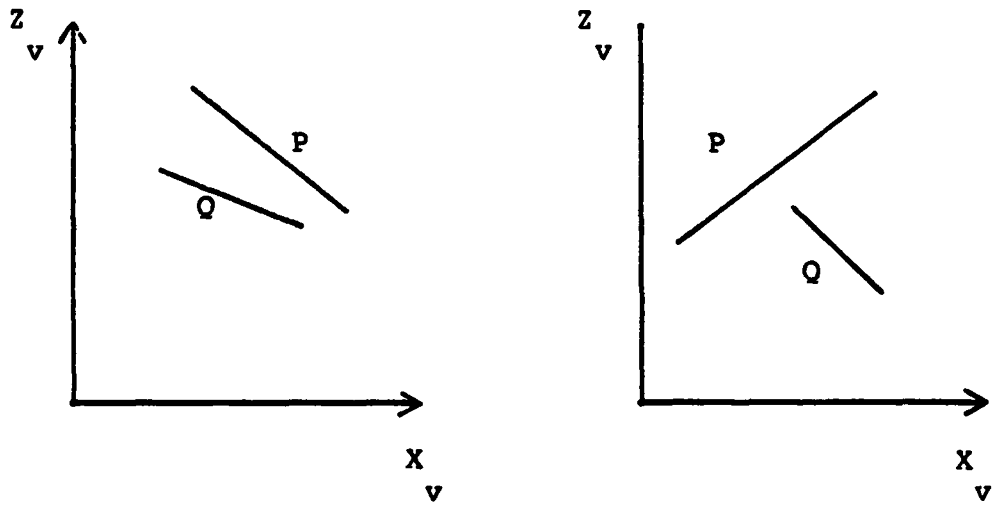


Fig 5. Plane Equation tests

If the extent tests and the plane equation tests fail to resolve the question of whether P obscures Q, there is a sixth test suggested by Newell, Newell, and Sancha. This test involves checking the projection of the polygons onto the screen to see if the polygon images overlap. To accomplish this task, all edges of each polygon are converted to screen coordinates and then the parametric line equation of the resulting lines on the screen are computed. We then find the point of intersection of each line in P and all lines in Q_i. If lines intersect along the line segments of either polygon they indeed overlap on the screen and P may obscure part of Q_i (Figure 6a). Additionally, if any line in P when extended to infinity crosses an odd number of Q_i lines, then at least one of the endpoints of the line of P is inside the polygon of Q_i and overlap on the screen will occur (Figure 6b).

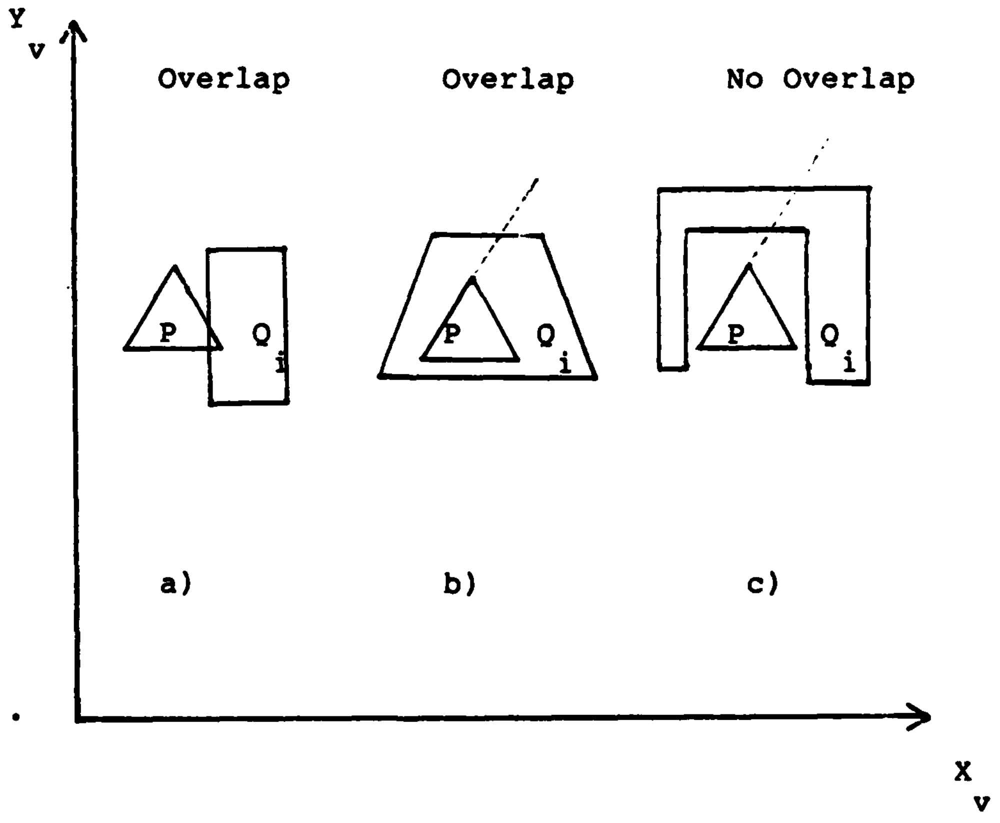


Fig 6. Scan Overlap

Determining the intersection of a line in P and a line of Q_i is based upon elementary geometry and the use of the parametric equation of a line.

$$x = x_1 + t(x_2 - x_1)$$
$$y = y_1 + t(y_2 - y_2)$$

Values of $t > 1$ indicate that a point is on the positive extension of the line segment from $(x_2 y_2)$ toward infinity. Similarly, values of $t < 0$ indicate that a point is on the negative extention of the directed line from $(x_1 y_1)$ to $(x_2 y_2)$. This is important for our testing because the projection of each edge of a polygon on the screen can be considered a directed line from some $(x_1 y_1)$ to some $(x_2 y_2)$. To calculate the intersection of two lines (one from P, the other from Q_i) we observe that unless the lines are parallel, they will intersect at some point x_{int} y_{int}. At this point, the x of the "P" line will equal the x of the "Q_i" line and similarly the y's of both P and Q_i will be the same. Recalling the above discussion, it is easy to see that we can find the values of the t parameters of both the P line and Q_i line at the intersection point by setting the parametric line equations for x and y to be equal and solve for one of the t values. For example :

Let x,y be the intersection point on the line of P.

Let x',y' be the intersection point on the line of Q_i.

Then:

$$x,y = x',y'$$

$$x_1 + t(x_2 - x_1) = x_1' + t'(x_2' - x_1')$$
$$y_1 + t(y_2 - y_1) = y_1' + t'(y_2' - y_1')$$

Solving for t in both equations we find that :

$$t = \frac{x_1' - x_1 + t'(x_2' - x_1')}{(x_2 - x_1)} = \frac{y_1' - y_1 + t'(y_2' - y_1')}{(y_2 - y_1)} \qquad (1)$$

Solving the second equality for t' gives:

$$\therefore \quad t' = \frac{(x_1 - x_1')(y_2 - y_1) + (y_1' - y_1)(x_2 - x_1)}{(x_2' - x_1')(y_2 - y_1) - (y_2' - y_1')(x_2 - x_1)}$$

t' is the parameter of where the line of the projected edge of P (x_1y_1 --> x_2y_2) intersects the line of the projected edge of Q_i ($x_1'y_1'$ --> $x_2'y_2'$). To find t, the parameter of where the Q_i edge intersects the P edge, we simply use the t' just calculated and substitute it into equation 1. If the values of both t' and t are between 0 and 1, then the edges of Q_i and P intersect and overlap will occur.

Unfortunately, one polygon may "surround" another (Figure 6b). In this case none of the edges will actually intersect, although certainly the inner polygon is overlapping the area of the larger outer polygon. To handle this case, the algorithm counts the number of intersections of the edges of Q_i by a line that extends to infinity from the projected line segment of P. This just requires counting the number of times t (the parameter for where along the P edge the intersection occurs) is greater than 1 and at the same time t' is between 0 and 1 (the Q_i edge was intersected). If, when all the edges of Q_i are checked against P, this count is odd; then part of P is inside Q_i and overlap has occurred. Because Q_i may be "inside" P, a similar count is kept for the number of times an extended Q_i edge intersects the edges of P. Again, if this count is odd when all of the P edges have been processed, then overlap has occurred.

At this point in the algorithm, we have not been able to prove that P will not obscure Q_i and although the ordering may be correct, the calculations to prove one way or the other were deemed by the originators to be too costly. However, Newell,Newell, and Sancha found during their evaluation of the algorithm that it was worth the effort at this point to attempt to reorder the placement of Q_i. Therefore, the algorithm performs the reverse of plane equation tests 4 and 5 to see if Q_i is wholly behind P or P wholly in front of Q_i. Although these conditions could have been detected during the initial plane equation tests, their calculations have been delayed until now to prevent unnecessary calculations. For example, if the first vertex tested in the first plane test indicates that not all of P is behind Q_i (the first vertex of P is found to be in front of Q_i), then the rest of the verticies of P could be examined to see if they also were in front of Q_i. However, if the next plane test (all of Q_i is in front of P) successfully resolves the ambiguity, the calculations done to prove the reverse of the first test would have been wasted.

If, as a result of the reverse tests, it is found that Q_i is wholly behind P or P wholly in front of Q_i, then Q_i is out of order and should be scan converted before P. If this swap is needed, Q_i is placed in the linked list behind the present P, marked that it has moved, and testing begins as before using Q_i as the new P for comparison purposes.

The flag signifying that the polygon is moved is needed because conditions can arise that cause cyclic overlap of polygons. This occurs when parts of polygons overlap in such a way that no matter what order they are scan converted, a correct image will not be created. In this case, swapping would be attempted by the algorithm indefinitely to resolve the ambiguity. Therefore, swapping is permitted only once before polygon splitting will be accomplished (there is one exception to this rule which is described a little later on).

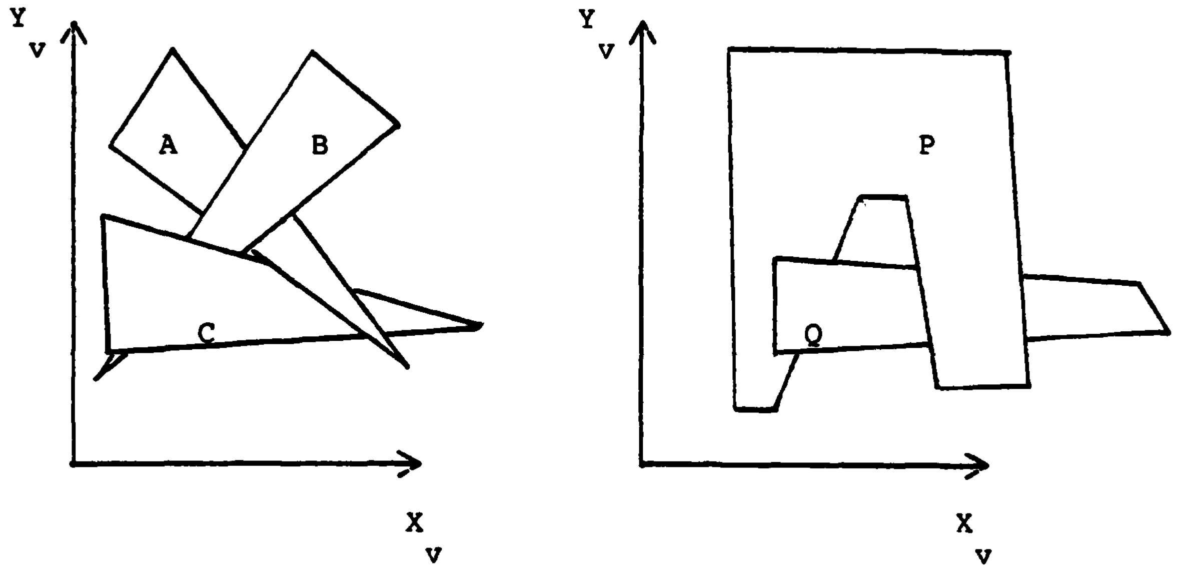
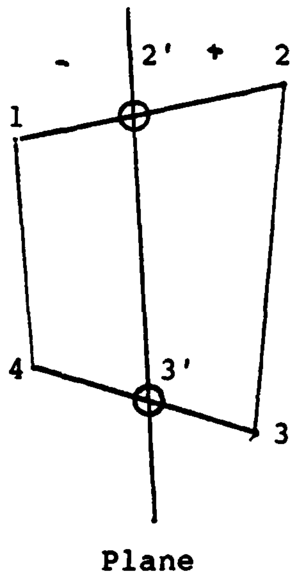


Fig 7. Cyclic Overlap

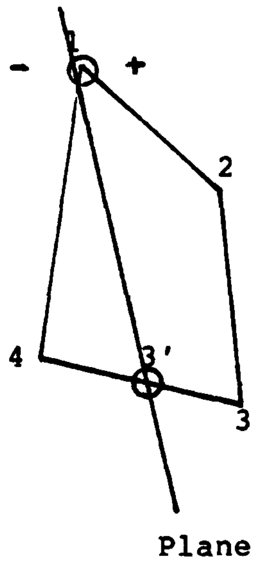
Polygon Splitting

If this reordering test fails, then the algorithm splits one of the two polygons using a method developed by Sutherland and Hodgman[19]. This method involves substitution of each vertex of polygon A into the plane equation of polygon B (same as test for whether a polygon was in front or behind the other). Points on the positive side of the plane will be placed in one list of new vertices and points on the negative side will be placed in another. Points residing on the splitting plane are placed in both lists as well as the points created when the polygon edge intersects the splitting plane.



vertex list

-	+
1	2'
2'	2
3'	3
4	3'



vertex list

-	+
1	1
3'	2
4	3
	3'

Fig 8. Polygon Splitting

The splitting algorithm proceeds as follows:

1) Calculate the value of the test vertex of the polygon substituted into the plane equation of the splitting plane. If this is the first vertex save this value for later.

2) If the test point is on the plane, place its coordinates into both the positive and negative side lists.

3) If the test point does not lie on the plane, then there is a possibility that the line from the last vertex to this vertex has crossed the plane and further checking must be done.

a) If the signs of the plane value calculated during the last iteration and for this point are different then:

i) Find the intersection of the edge of the polygon and the splitting plane.

ii) Place this intersection point in both the positive and negative side lists.

b) Place the test vertex into the positive or negative vertex list based on the sign of the plane value.

4) Save the plane value calculated for this test point for the next iteration, update the counters for the edge under consideration and go back to 1 until all vertices have been checked.

5) "Close" the polygon by examining the edge that runs from the last vertex in the list to the first.

a) If the first vertex or the last vertex is on the splitting plane, then the lists are complete; nothing else needs to be accomplished.

b) If the first or last vertices are not on the splitting plane then there is a possibility that the line from the last vertex to the first vertex crosses the splitting plane and further testing is needed.

If the signs of the plane values of the last vertex and the first vertex are different, then :

i) Find the intersection point of the edge of the polygon and the splitting plane.

ii) Place this intersection point in both lists of vertices.

After this closing step, there are two separate lists of polygon vertices that define the parts of the original polygon found on the positive and negative sides of the splitting plane.

The one major calculation involved with this splitting process is the determination of the intersection point of polygon edges and the splitting plane. To find this point, geometry and and the parametric equation of a line are again used in the solution. In addition, the parametric equation of the splitting plane is used along with the observation that the intersection of the edge and plane must satisfy the parametric line equation for the NDC x,y, and z edge of the polygon and also be in the locus of points that satisfy the splitting plane equation.

This point can be expressed in parametric form as:

$$x = x_1 + t(x_2 - x_1)$$
$$y = y_1 + t(y_2 - y_1)$$
$$z = z_1 + t(z_2 - z_1)$$

The equation of the plane can be expressed as $aX + bY + cZ + d = 0$. By substituting the x,y,z of the intersection point above into the plane equation we find :

$$a[x_1 + t(x_2 - x_1)] + b[y_1 + t(y_2 - y_1)] + c[z_1 + t(z_2 - z_1)] + d = 0$$

Solving for t yields:

$$t = \frac{-(ax_1 + by_1 + cz_1 + d)}{a(x_2 - x_1) + b(y_2 - y_1) + c(z_2 - z_1)}$$

This is a very fortunate result, because the numerator (without the minus sign) was calculated during the previous iteration as the plane value. Additionally, the denominator is the plane value calculated this iteration minus the previous iteration's plane value. We simply solve for t and then use this value to find the x,y,z of intersection (e.g. $x_{intersection} := x_1 + t(x_2 - x_1)$).

This value of t also has other uses. The CORE allows the user to define color values associated with each vertex of the polygon. By varying the color intensities at the endpoints, the user can implement shading "on top of" the CORE. Because we are splitting a polygon, new color values will have to be made for the new points in the vertex list which were caused by the intersection with the splitting plane. Otherwise, the shading information will be lost and improper picture generation will result.

The value of t is simply the percentage of the distance along the edge from $x_1 y_1 z_1$ to $x_2 y_2 z_2$ where the intersection has occurred. If we consider the red, green, and blue that make up the color value at each endpoint, we find that these too can be expressed in terms similar to that of the parametric value of x, y, and z. For example consider the red color value as red_intersection := red_1 + $t(red_2 - red_1)$. Therefore, we can linearly interpolate the color index information at the original vertices by simply substituting t into the red, green, and blue equivalence to the parametric line equations and storing the values calculated into the color index array.

There are two possible ways in which a polygon can be split. First, a test is made to see if parts of P lie on both sides of Q_i. This test is made by substituting the polygon vertices into the plane equation of Q_i. To save time, all of the plane values calculated during the plane equation tests were saved, so that only if needed will new plane equation calculations be performed. If parts of P do lie on both sides of Q_i, the P polygon is divided by the plane of Q_i. If this fails, we determine if points in Q_i lie on both sides of P. If so, the Q_i polygon is divided by the plane of P.

Recall that under normal conditions if a reordering is attempted on Q_i but Q_i has been moved, we will split one of the polygons. However, there is one exception to this rule. This exception occurs when a Q_i polygon has been moved on a previous swap and because of some other ambiguity is now on the wrong side of P. In addition, P and Q_i do not have points on both sides of each other, so the real problem is that they are simply reversed in the list and Q_i really should be scan converted before P. The original algorithm would at this point divide the P polygon in the hope that by dividing P the ambiguity would be eliminated. However, in cases used to test the algorithm, it was found that this type of resolving the ambiguity frequently required dividing P down to the size of a pixel, once a polygon that was moved got out of order. Therefore, the algorithm developed during the research eliminated this type of polygon splitting altogether and allowed polygons that had been moved to be swapped under the condition that points of both polygons were only on one side of the other (the wrong side).

Placement of Polygon Fragments

Placement of the splitting polygon and the two halves of the split polygon into the linked list is performed in a manner somewhat different than the method described in Newell, Newell, and Sancha's original paper. We propose that the P polygon is at a position in the list that up to this point best approximates its relative position to surrounding polygons. Therefore, if P is split by Q_i, then the original part of P on the back (negative) side of Q_i should remain in place and in fact all of the polygons up to and including Q_i in the set of Q that overlap P in z extent are correctly ordered. Therefore, I place the negative vertex list created during the polygon splitting process into the original P data structure, place the positive vertex list into a new hidden surface node, insert this new node into the hidden surface linked list sorted by z on the front side of Q_i (ignoring moved polygons), and set the Q_i pointer to the next polygon toward the observer. Thus, the part of P on the negative side of Q_i remains as the testing polygon, Q_i remains in position, and the new positive side polygon fragment is placed into a first guess position in z so that it will be scan converted after Q_i.

When the polygon Q_i is split by P, a different placement of the fragments is needed. The new part of Q_i that was on the positive side of P is inserted into the hidden surface linked list on the positive side of P, sorted by z. However, the negative side of Q_i must be placed directly "behind" P in the list and then must become the new P for testing. This movement behind P causes the z sorting values to be incorrect and therefore requires that Q be marked as moved. As a last step, the Q_i pointer is set to point to the polygon next in the list after the old P. When these steps are complete, the negative part of Q_i is the new test polygon (P), the old P remains in position, and the new Q_i is placed on the positive side of P, sorted by z.

Recall that at hidden surface initialization time we took the NDC points, performed the image transformation and stored the new image transformed coordinates in the hidden surface data structure. During the resolving phase, if we split a polygon, we split the polygon in the image transformed coordinates and did not calculate the NDC coordinates that coorespond to the new split polygon coordinates. Instead we have intentionally delayed "binding" the NDC equivalent coordinates to the split polygon coordinates until the polygon has been completely resolved. This late binding prevents unnecessary calculations on polygons that may have to be split more than once and as required by the CORE, allows the user to perform future image transformations on the original segment he defined, no matter how many times the polygons within the segment were split. This calculation of the NDC space equivalent points is done by calculating the inverse matrix of the image transformation for the primitive that was split and then multiplying each point in the hidden surface structure by this 4X4 matrix to arrive at the proper NDC values. These values are then stored into the output primitive data structures for future batches of updates.

Resolver Pseudo Code

To aid the reader in understanding the entire resolving process, the following pseudo code is presented:

```
RESOLVER:
 INITIALIZE P,$Q_i$
 WHILE PRIMITIVES REMAIN TO BE RESOLVED DO
   BEGIN
    WHILE $Q_i$ MAX Z > P MIN Z DO
      IF P,$Q_i$ X OR Y EXTENTS DO NOT OVERLAP THEN
         $Q_i := Q_{i+1}$
      ELSE IF P BEHIND $Q_i$ THEN
         $Q_i := Q_{i+1}$
      ELSE IF $Q_i$ IN FRONT OF P THEN
         $Q_i := Q_{i+1}$
```

```
ELSE IF NO SCREEN OVERLAP P,$Q_i$ THEN
   $Q_i$ := $Q_{i+1}$
ELSE IF $Q_i$ MOVED THEN
     IF POINTS OF P ON BOTH SIDES OF $Q_i$ THEN
        SPLIT P BY PLANE OF $Q_i$
     ELSE IF POINTS OF $Q_i$ ON BOTH SIDES OF P THEN
        SPLIT $Q_i$ BY PLANE OF P
     ELSE
       BEGIN
         MOVE $Q_i$ BEHIND P;
         P := $Q_i$;
       END
ELSE IF P IN FRONT OF $Q_i$ OR $Q_i$ BEHIND P THEN
  BEGIN
      MOVE $Q_i$ BEHIND P;
      P := $Q_i$;
  END
ELSE
  IF POINTS OF P ON BOTH SIDES OF $Q_i$ THEN
     SPLIT P BY PLANE OF $Q_i$
  ELSE IF POINTS OF $Q_i$ ON BOTH SIDES OF P THEN
     SPLIT $Q_i$ BY PLANE OF P
ELSE
  ERROR TRAP;


DRAW RESOLVED POLYGON;
IF POLYGON WAS SPLIT THEN
```

```
        CALCULATE NDC POINTS;
        P := P + 1;
        Q  := P + 1;
         i
     END;
```

As a result of scan converting all the polygon faces in order from back to front,this algorithm will eventually create a correct image. At this point, one may conclude that we can conceivably overwrite parts of the screen many times, and in fact we may perform very expensive polygon splitting computations on polygons that are eventually overwritten. This can occur unfortunately, but the tests are purposely ordered in increasing complexity and the algorithm spends little time in the more complicated polygon splitting portion. In fact, Newell, Newell, and Sancha point out that the time taken for ordering faces "is largely dependent on how good the initial ordering is. If few faces intersect this time is usually short, but cases can arise where it becomes significant."[13:447] As it turns out, the bottleneck to this algorithm is the scan conversion time needed to paint each polygon face. "..Clearly this is a part of the algorithm that can be isolated and implemented in hardware."[13:447]

This chapter has discussed in detail the hidden surface algorithm implemented during this research. The next chapter discusses its usefulness.

V Analysis of the Final system

This chapter is devoted to describing the graphics system created by the merger of the Flight Dynamics Laboratory's microcomputers, the University of Pennsylvania's CORE software package, and the Hidden Surface Removal algorithm. The first section describes the serious limitations on the system caused by the Pascal MT+86 compiler and linker. The second section briefly outlines the changes made to the University of Pennsylvania software to reduce the data space and computation time used prior to the invocation of the hidden surface routines. Finally, this chapter closes with some conclusions reached about the design of the present implementation and the overall performance of the algorithm.

Hardware and Support System Capabilities

The single most devastating and complicating limitation experienced during this research was the 64K byte limitation on program area length. As mentioned earlier this caused the need for program overlays and still only gave the user 18K for application program development. This is unacceptable for major project use.

In the future, the Flight Dynamics Laboratory wants to use this system to create complex scenes from a data base containing many points and geometric figures. The present CORE package is very general in nature and allows the user to create either perspective or orthogonal 3-D views of objects. Since the Lab wants perspective views only, the main routines could be shortened by eliminating the checks for orthogonal views and associated processing. This would eliminate rather large chunks of code in the routines used to create the viewing transformation matrix and perform the clipping operation. Smaller gains could also be realized in the hidden surface procedures and the routine used to map the view volume to the viewport. Additionally, the initialization routines could be tailored to to set the initial viewing parameters to an initial 3-D system vs the standard 2-D viewing system parameter defaults of the CORE. Finally, many of the CORE error checking features of the software could be eliminated such as the check for system initialization in all of the user callable routines.

This approach would result in a system that would not be CORE standard, would not be for general application, and would not protect itself from incorrect user input. This would also defeat one of the main goals of the CORE, namely transportability of the application programs from one facility to another. Although undesirable from an academic point of view, this approach would eliminate several routines from the device independent overlay, and if continued into the device dependent overlay area by eliminating the routines used to support marker, polymarker, and the escape calls of the original package, would save significant room for application program use.

Another possibility was discovered during a telephone conversation with software personnel of Digital Research. They plan within the year to release a new version of their PASCAL MT+86 compiler that would remove the 64K program area restriction. This anticipated system would also be compatible with their C and FORTRAN compilers so that procedures created under PASCAL,FORTRAN, or C could be linked together to form a system. The change from the present overlay defined CORE package to such a system would require minimal software change, but cannot be counted upon because of the lack of a firm date of this compiler version and the need for graphics now.

Of course, the laboratory could also search for a different compiler for the 8086 that did not have the 64K limitation. If one were found, they would then have to convert the software from the present MT+86 format to the Pascal of the new compiler. This conversion effort could take quite a few manhours if the conversion from the VAX Pascal to MT+86 is any indication.

Alternatively, the laboratory could purchase a more advanced Graphics display device that performs much or all of the CORE functions. By offloading the functions to the device, the present CORE user routines could be modified to create the interface between the 8086 and the device. Additionally, the applications programs developed during this research could be utilized for the checkout of the device. Depending on the amount of support given by the software and hardware "on board" , the 8086/8087 user program area could be substantially increased.

Other problems were found in the system support software that are also worthy of mention. First, the Pascal NEW and DISPOSE procedures of MT+86 do not work properly. As long as the user asks for "new" memory the system properly allocates memory space and provides an address for the user. However, whenever DISPOSE is called, appearently the linked list of available memory is corrupted and a subsequent call to NEW will cause an infinite loop. This problem prevented returning memory area to free space taken by primitives that were completely clipped by the CLIPPER subroutine. More importantly, this complication prevented the deletion of created segments by the user and would not allow the movement of an eyepoint through a data base of segments.

An overlay manager problem was also encountered during the research. Pascal "REAL" functions would not return proper values in an overlay environment. Finding this error was difficult because of the correctness of the Pascal code involved and the natural suspicion that incorrect overlay external declarations were being used. However, after displaying values computed inside and passed outside of real functions, the error was detected and reported to the compiler manufacturer.

Inclusion of the functions into the driver procedures was accomplished by either making the functions into procedures with a return variable (used when more than one procedure called the function) or by including the function into the declaration part of the calling procedure. Because a function would work when included within a calling module, there is reason to believe that the overlay manager somehow corrupts the parameter stack when calling real functions in other modules.

During the development of the overlay areas, it was found that symbol table space was needed behind the last procedure in the overlay for the overlay manager. The overlay manager used this symbol table to lookup the address of a called procedure within the overlay. Whenever the size of code within the overlay got within approximately 3K bytes of the allotted area for the overlay, the symbol table could be overwritten at read-in time causing the overlay manager to fail when trying to find the procedure name. To counter this problem, the MT+86 system had an undocumented system utility called STRIP. With this utility, it was possible to throw away unneeded symbols for variables and compact the symbol table considerably. In addition, a record normally containing all zeros was eliminated at read-in time which again saved space. Unfortunately, this utility took around 5 minutes to run each time I changed a procedure in an overlay because STRIP required the manual entry of all procedure names within the overlay (overlay 3 had 55 procedures).

Finally, the Scion graphics display processor interface did not provide real time picture generation. To draw polygons, the University of Pennsylvania graphics package contained a procedure called PPOLY which took a list of verticies and performed scan line conversion of the polygon. In other words, it figured out a list of lines in y on the screen that when displayed would create the filled polygon on the screen. In one test, a rectangle that covered the entire screen was completed in 17 seconds.

University of Pennsylvania Output Pipeline

The original software package delivered by the University of Pennsylvania was excellent. It was very well documented, well designed, and was clearly made with the design concepts and functional specifications of the CORE standard. Its modularity allowed easy incorporation of the hidden surface routines and enabled quick determination of faulty procedures during debugging. However, I made a few basic changes to the output pipeline to not only save space but reduce computations on the data, including the development of a new data representation of polygons, the incorporation of new perspective viewing procedures, and the reduction of the error printout routine program size.

The clipping algorithm used in the package is not a polygon clipper, but rather a 3-D line clipper. To allow polygons to be clipped to the view volume, the University of Pennsylvania stored the list of polygon verticies passed by the user, as a list of line segments with the beginning and ending points of a line corresponding to an edge of a polygon. Thus for a triangle of points A,B, and C, six coordinates would be stored representing the 3 sides (e.g. points arranged A,B,B,C,C,A representing the edges A-->B,B-->C,C-->A). This double representation of points caused both twice the space requirements and twice the computation of a simple list of polygon verticies.

Each point stored in the primitive data structure was originally multiplied by a 4x4 viewing transformation matrix, clipped by setting values outside the viewing volume to a negative number, then regardless of the clipping, transformed to the viewport by the window to viewport maping routine. Inside the actual drawing routines, a check was finally made to determine if a point was outside the view volume. If it was not, it was sent to the polygon scan conversion routine after the final user defined image transformation. Additionally, each point could be multiplied by a 4x4 user instancing transformation, and if perspective views were wanted, a perspective division was necessary before the viewport mapping. Because polygon points were stored twice, all of the processing including the user instancing transformation, viewing transformation, perspective division, and window to viewport mapping were accomplished twice per point, which was deemed an unacceptable waste of CPU time.

Therefore, the user callable polygon routine (POLYA3) was modified to store the list of verticies exactly as given by the user, one point per vertex. Thus the user and viewing transformations were accomplished only once per point. The clipping routine was then modified to create a temporary area that placed the transformed points in the original double storage format. The main clipping procedure remained the same and then a procedure was added to the output side of clipper to eliminate points culled during the clipping process. Thus, only relevant points were passed down the pipeline for the perspective division and window to viewport mapping routines. If the whole primitive was clipped, a call to DISPOSE to free the primitive data area was incorporated into the code, but as mentioned earlier because DISPOSE was not functioning properly, this call was commented out. Overall, in addition to the computation savings, this polygon representation change halved the space needed to store the coordinates.

Unfortunately, the original package sent early in the research had some problems with perspective views of scenes. The orthogonal worked properly, but because the laboratory needed perspective views, a dilemma was created. Professor Badler of the University of Pennsylvania was more than helpful and near the end of the research sent to us a working perspective CORE package. Because the orthoganal was working, I was able to work out the major bugs in my subroutines before the update arrived.

Incorporation of the new version was never fully accomplished because it was at this critical hour that the problems with real functions started as a result of the overlays. Early tests with the changes indicated that the y coordinates being generated were improper and after some time insuring that the new code was properly inserted, these changes had to be abandoned. Fortunately, in earlier course work perspective routines had been developed and after these were incorporated, research continued.

Finally, the original University of Pennsylvania sortware had an extensive error printout routine that had textual descriptions of errors found during operation of the CORE. Although fantastic for debugging, this routine (because of the many output strings) took up too much room for use on our system. Therefore, its function was reduced to merely printing out an error number and procedure name. When errors occured, the original listing of the procedure had to be consulted to find out what error had occured. Although more time consuming, this was the only feasible alternative.

The Algorithm

The Newell, Newell, and Sancha approach to rendering correct images on the screen can be characterized by three distinct phases of operation. The first, the initial sorting phase, takes a list of polygons and orders them by their maximum z . Second, a resolving step takes this initial estimate of the correct "painting" order and eliminates conflicts between objects. Finally, after a polygons's conflicts have been resolved, a process is needed to convert the description of the polygon into a physical display on some device. This section will comment on the results of my research on the first two functions of the algorithm (many hardware devices perform the last function).

Sorting

My initial attempt at creating the data structures and procedures to operate upon them centered around simplicity of function so that an acceptable algorithm testbed could be developed in minimal time. One area that truely suffered from this approach was the initial sorting step. In retrospect, this part of the implementation needs serious thought and further research to produce a better first guess ordering of the polygons.

First of all, a wrong assumption was made that the user would most probably add polygons in groups that would be close to one another in a linked list. Therefore, the SORTER routine created a doubly linked list structure and maintained a pointer to the the last polygon inserted into the list. If the merging entry had a maximum z greater than the last entry, SORTER searched backwards in the list (further from the eye) until a proper place was found for the merging entry. In the same way, upon finding a merging entry with a maximum z less than the last entry, SORTER went forward until a place in the list was found. This search was known not to be as efficient as a binary search, but it was rationalized that because of the clustering effect of incoming entries, it would be acceptable.

Long searches were found however, when objects were defined front to back. This resulted from having to search through groups of polygons that shared the same z, such as groups of terrain polygons. One reasonable approach to solving this problem would be to store the merging polygon entries in a binary tree structure rather than the present singlely linked list. By performing an inorder traversal of the tree, the doubly linked list needed during the resolving phase could quickly be made for the first frame. Subsequent batches of inserts could also be ordered in a tree and then a one time inorder traversal of the tree while scanning from back to front in the hidden surface linked list would place the incoming polygons in their proper positions. This would make a tremendous savings of time in the first sort, and may indeed save time in subsequent batches of updates.

As noted by [20:41], additional information as to the sorted x,y position of polygons could be added to the hidden surface structure to cut down on these unneeded comparisons. In addition, a totally different scan line algorithm could be made using a y bucket sort for each scan line followed by the Newell, Newell, and Sancha resolving process for z ambiguities, terminating in a final x bubble sort[20:41,2]. This new algorithm could use large chunks of the present algorithm but would definitely need new data structures.

Resolving

The following table contains a profile of the number of times the major algorithm tests were peformed on four different scenes (Figure 9):

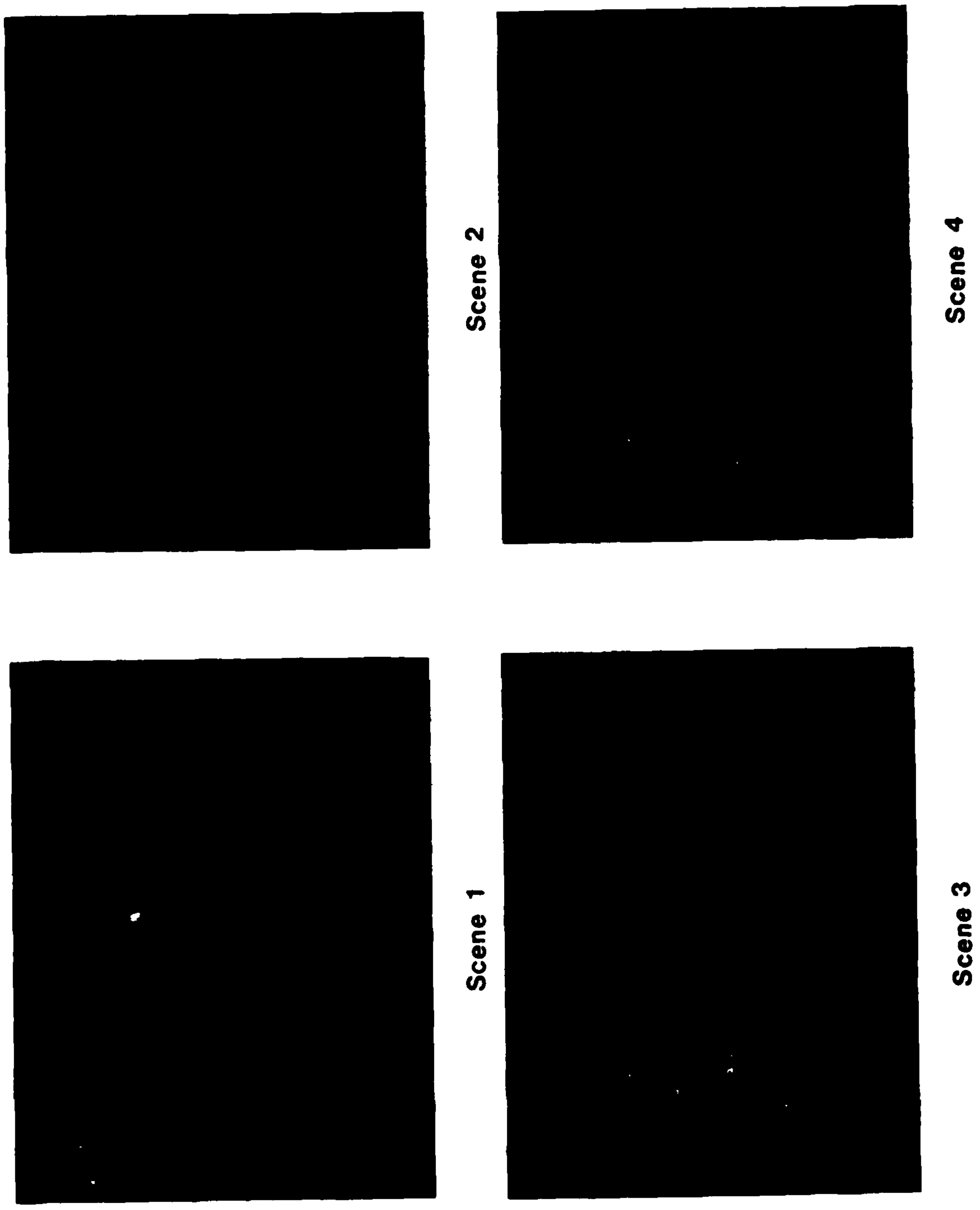


Fig 9. Test Scenes

Table I

Algorithm Performance In Four Scenes

	1	2	3	4
Beginning # of polygons	34	35	236	236
ending # of polygons	34	64	237	278
# of xy extent tests	279	675	3713	4562
# of P behind Q tests	54	208	312	540
# of Q in front P tests	26	121	59	341
# screen overlap tests	8	51	27	189
# reoderings attempted	6	44	12	80
# swaps performed	6	15	11	43
# polygon splits	0	29	1	42

It is obvious that the algorithm performs many more tests for x,y extent than any other. This was expected and in fact hoped for. However, this large number also reveals that each polygon is being compared against polygons that could have been eliminated by some previous x or y sorting scheme. As noted by [20:41], additional information as to the sorted x,y position of polygons could be added to the hidden surface structure to cut down on these unneeded comparisons. In addition, a totally different scan line algorithm could be made using a y bucket sort for each scan line followed by the Newell, Newell, and Sancha resolving process for z ambiguities, terminating in a final x bubble sort[20:41,2]. This new algorithm could use large chunks of the present algorithm but would definitely need new data structures.

Additionally, it can be seen that there is a significant increase in the number of tests required to resolve a scene with overlaping polygons vs a scene without overlap. This is due in part to the fact that there are more polygons that have to be resolved as well as the fact that whenever we have to split Q_i by P, we will have to perform the x,y extent and other tests on the half of Q_i placed behind P as well as reperform the tests for P.

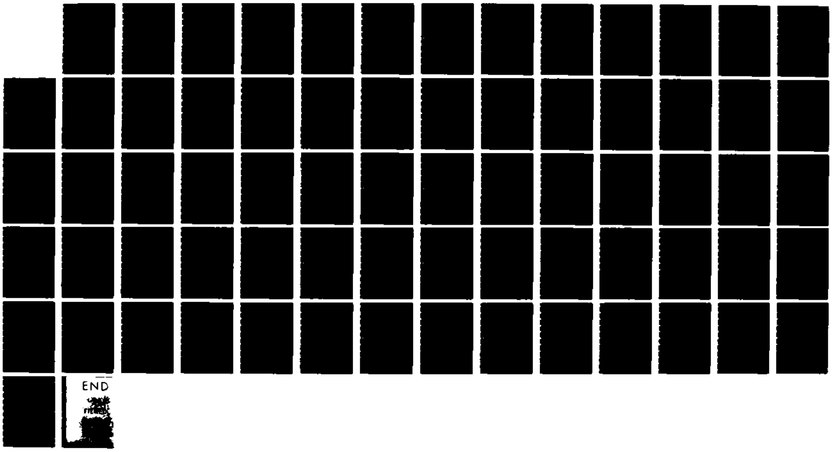
The plane equation tests for P behind Q_i and Q_i in front of P drastically reduce the amount of unresolved polygons with their effectiveness highly dependant upon the existance of penetrating polygons. In the two simple scenes (1 and 3), they reduce the number of unresolved polygons by at least a factor of 7 and in the overlapping scenes, by factors of around 4 and 3. This is of course explained by the fact that penetrating polygons will have points on both sides of the plane. Because a significant amount of processing is done to resolve polygons using the plane equations (3 floating point multiplications and additions per vertex of P and Q_i if it takes both tests to resolve the ambiguity) this is definitely one routine that should be coded in assembly language. Because the indexing is straight forward and the plane equation coefficients remain the same from iteration to iteration, an assembly level programmer should be able to save many steps needed by the more general Pascal compiled version.

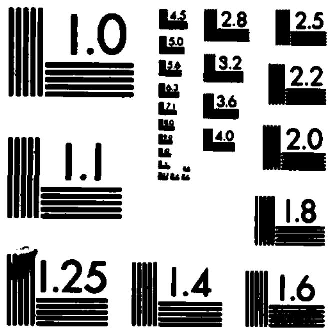
The screen overlap test is expensive. First from a computational point of view, it takes 5 floating point multiplications, 2 floating point divisions, 2 floating point additions, and 1 floating point subtraction to compare one P edge to one Q_i edge. Considering that a total of P edges X Q_i edges X (5 multiplications, 2 divisions, 2 additions, and a subtraction) calculations will be needed to prove P and Q_i don't overlap, it is easy to see that a search for an easier scan overlap test is in order. Indeed, the only thing that prevents this test from having a larger impact is the fact that only a few ambiguities remain. Of course at a minimum this should be placed into assembly language.

During research, it was noted that although swapping seemed like a very cost effective way to resolve ambiguity, it had a higher cost than expected. First of course, by moving a polygon in the list, complications arise because we ignore "moved" polygons in our sorting and placement routines. Conceivably, we could reorder many polygons over time when the user does many batches of updates with a different view point. Thus the first guess approximation sort would become less and less effective. This is definitely an item of further research.

Second, it was noted that when swapping occurs, many

AD-A138 239 PLACING HIDDEN SURFACE REMOVAL WITHIN THE CORE GRAPHICS 2/2
STANDARD: AN EXAMPLE(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. T S WAILES
UNCLASSIFIED 07 DEC 83 AFIT/GCS/MA/83D-9 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

ambiguity checking steps are going to be repeated. Recall that the swapped polygon is placed behind the present P in the list and becomes the new P test polygon. If this arrangement results in the new P being resolved, then the P pointer is advanced forward to the old P and its testing begins once again. All the comparisons up to the old swapped Q_i position will be repeated and if many polygons overlap P in z, this can be significant.

Polygon splitting is expensive both in computation time and data structure space. First the equivalent of the plane test is going to be performed on the vertices of the polygon being split. Presently, the algorithm does not take advantage of this fact, and any plane value computation time taken in the main resolver program will be repeated in the polygon splitting procedure. Additionally, the calculation of the intersection of the polygon edge and splitting plane is going to take 7 floating point multiplications, 1 floating point division, and 6 floating point additions for each time an edge traverses from one side of the plane to the other.

These computation times appear to be less than that caused by checking for scan overlap, however remember that each time we split a polygon, we must perform the image transformtion, and completely resolve it; perhaps involving the comparison of many other polygons. Finally, each additional polygon created requires us to make not only another hidden surface node but another primitive "host" node as well. If the user creates a large scene that has many penetrating polygons, the additional space requirements may exceed his available memory. Generality does not come without cost.

Computational Cost of the Algorithm

Perhaps, the best way to model the computational cost and performance of this algorithm is to ponder the way polygons are culled from consideration. All polygons that overlap the test polygon will at least be tested in the extent tests. This isolates the test polygon's further testing to those polygons lying in a cube formed by the x,y and z extent box. The first plane test culls those polygons on the front side of the test polygon only and reduces the possible area of conflict to the "back" side of the extent cube. The second plane test divides the extent cube by the plane of Q_i and if all of P resides in the negative side of that portion of the box then the ambiguity is resolved. The scan overlap test basically is assuming that the polygons are somehow out of order, but if they can be scan coverted without disturbing the finished image, the present ordering is okay. The test for swapping again assumes that they are out of order and attempts to prove the same. If they cannot be swapped, then a polygon splitting will have to be made to resolve the conflict (almost always caused by penetrating polygons).

Thus, it can be said that this algorithm is going to perform relative to the geometric complexity of the generated scene not its visual complexity. If the user specifies a scene consisting of 1000 polygons with the last polygon overwriting all the others, it will not take advantage of the visual coherence of the scene (e.g. that the last polygon is the only one that needs to be displayed). Instead, it will systematically resolve the scene back to front until all polygons have been displayed.

Now that the graphics system and hidden surface algorithm have been analyzed, it is time to summarize the proposals mentioned in this thesis.

VI Conclusions and Recommendations

This paper has documented the design and development of a hidden surface algorithm and its integration into the CORE graphics standard. Many improvements to the system were presented throughout the work and therefore the purpose of this chapter is to place all these recommendations and conclusions in one area for those interested in pursuing research using this algorithm and those wanting to improve its present implementaion.

First, this chapter outlines the changes needed to make the current software package completely conform to the CORE graphics standard (recall that in chapter one the problem was limited to creating a polygon processing algorithm for raster type devices). Then a section describes the areas of further research that could conceivably result in a more efficient algorithm implementation. Finally, a general course of action is recommended for the Flight Dynamics Laboratory.

Making the Algorithm Standard

There are four basic limitations on the present implementation that make it non standard: it only supports raster type displays, it does not perform hidden line removal, it only presently resolves polygons, and it does not support all hidden surface levels of the CORE standard. This section will describe the needed changes to allow all of these improvements.

The simplest way to incorporate both support for calligraphic displays and hidden line removal would be to use the sorting and resolving steps of the present algorithm as a prelude to the Watkins or a similar type scan line algorithm. The only changes needed in the present routines would be the deletion of the call to the scan line converter in the main resolving subroutine and in its place the addition of code to save the priority number of each polygon. The present data structure would have to be changed to incorporate the priority number, and the Watkins y bucket sort linked list and active segment linked list data structures would have to be created. Of course the major work would be to design the actual algorithm that would use the results of the Newell, Newell, and Sancha algorithm to create the displays.

This approach could well be unacceptable because of the additional data structure space needed by the new routine and the additional computations on top of the Newell, Newell, and Sancha algorithm. An alternative to this approach will be discussed in the section on further research.

The changes needed for extending the algorithm to process the line, polyline, marker, polymarker, and text primitives simply involves the addition of logic in the main resolver routine. The marker and polymarker primitives would simply be treated as points in space. Using the depth sort process we would simply draw these objects at their proper depth priority and let objects closer to the eye overwrite them if needed (for the calligraphic type displays their priority order will determine if they are displayed). Lines will simply require additional checks to determine if intersections occur with polygons. If an intersection does occur the line will be divided at the point of intersection and the two line halves and the polygon will be ordered and inserted in the hidden surface linked list and output primitive data structures. Polylines will be considered as collections of individual line segments for this processing. Finally, text primitives will simply be treated as 3-D rectangular polygons that are defined by the extent box surrounding the strings of characters. Obviously, the plane equation tests and scan overlap tests would have to include different logic to process these other cases, but the logic required is straight forward and most of the algorithm would remain the same.

Perhaps, the easiest change to make is the logic to allow all four levels of hidden surface removal (none, temporal, explicit, and full). Level 1 simply means no effort is taken by the package to order the user's output primitives. To implement level 2, the present placement of primitives within the segment primitive linked list would require a slight modification.

Level 2 or temporal priority, requires that primitives are displayed in the same order as they were created by the user. The University of Pennsylvania CORE routines currently add primitives such that they end up in reverse order from that given by the user. By simply adding a pointer to the last primitive added to the presently open segment and using this pointer to insert primitives, level 2 could be easily accommodated.

Although the current implementation of the algorithm would support level 3, many of the present hidden surface initialization calculations are not needed in level 3. Therefore, it would be best to create an additional hidden surface sorting routine and, to save space, create a new binary tree structure. All the new sorting routine would have to do is maintain a binary tree structure that ordered the primitives by z (priority). After all incoming primitives were placed into the tree, an inorder traversal of the structure would produce the correct rendering of the scene. The tree data structure would only require the storage of the z (priority) given by the user, the image transformed coordinates, and three pointers (one back to the host primitive, two for the left and right branches of the tree structure). This would save the memory area presently taken by the x,y, and z extents, the plane equation coefficients, and the Boolean flags used in the present hidden surface node data structure.

Items of Future Research

During the the debugging of the algorithm, I was able to watch as the procedures resolved scene ambiguity and noted several items that need to be investigated further. This section discusses these items, first outlining the changes that would make the present implementation more efficient and then discussing the areas in which new ideas are necessary.

As mentioned earlier one important improvement would be the incorporation of a binary tree structure to order incoming primitives before the main resolving process begins. The initial ordering is a very important part of the algorithm and on the first batch of updates can be a significant component of the time needed to create the image. Additionally, a reduction in the time needed to split polygons could be realized by taking advantage of the plane values already calculated as part of the plane equation and reordering tests. By making these available to the polygon splitting routine, the present duplication of calculations could be avoided. Finally, an improvement can be realized by placing the plane equation and scan overlap tests in assembly language. As a start, the Pascal MT+86 compiler provides an assembly code printout of each line as it is compiling. An assembly code programmer could easily use this printout as a first cut, and with proper knowledge of the data structures and the goal of each procedure, cut corners that a general purpose compiler simply cannot.

Further testing is needed to determine the effect of swapping during many batches of updates. It is conceivable that the number of moved polygons could quickly climb if the initial ordering of polygons is improper due to the ignoring of moved polygons when placing primitives into the linked list. This feeding of the problem upon itself could result in a list that spent a large part of its time swapping primitives. This could be detected simply by keeping track of the present number of swapped polygons, and one solution would be to resort the list by z after a threshold of moved polygons existed. This may not really be a problem, but it certainly must be examined.

The present scan overlap test takes quite a few calculations to prove that one polygon does not overlap another. A more computationally effective method could in itself be the topic of research as this is not the only hidden surface algorithm that uses this kind of surrounder test. Additionally, during research it was noted that swapping caused duplicate calculations because polygons up to the polygon causing the swap are recompared in the present implementation. If some way could be developed to save the set of polygons already compared such that when the swapped polygon was resolved the old P could continue where it left off, a number of comparisons could conceivably be saved.

The most exciting area of research, of course, would be to use the existing routines in a totally new way to come up with a new algorithm altogether. As mentioned before this new algorithm would combine the ideas of scan line algorithms and the Newell, Newell, and Sancha resolving process to make a hybrid algorithm that took advantage of the strong points of each. The new algorithm would first sort the primitives using a y bucket sort placing the primitives into buckets that corresponded to the y scan line they first appeared (going from top to bottom). We would then start the normal scan line type processing of creating spans of polygons. The present resolver procedure would be used to order the list in z as each primitive entered the active edge list.

> "... Because newly-entering faces are added to a list from the previous scan line which is already in order, depth coherence from scan line to scan line can be preserved. Newell's sort is thus simplified to finding an appropriate position in the priority list for each newly-entering face."
>
> "The final X sorting step would make full use of the scan-line coherence properties familiar in the other algorithms. X intercepts would be computed incrementally, and kept in X order by a bubble sort. When depth computations are required, a simple test of priority number

would suffice. One would make use of scan-line depth coherence by remembering which segments are visible from scan line to scan line and by repeating a visible segment if no edge crossing had occurred involving one of its visible edges. Note that penetration will have been resolved for entire faces during the Z sort process."[20:41,42]

Because the z resolving steps are already in place, the development of this proposed algorithm and its comparison in performance to the present algorithm would make an excellent topic for further research.

Recommendation

Further graphics research and future graphics application programs will be significantly hampered by the present 64K byte program area restriction of Pascal MT+86. Therefore, the most reasonable course of action would be to save the present system source programs on floppy discs (for future use if the size restriction is lifted) and then modify the programs to reduce their generality. Then as development continues, search for a multitude of software and hardware options to increase the application program area size. This will provide some graphics research in the near term (perhaps implementing some of the above mentioned suggestions) and if a suitable compiler, advanced display processor, or if a future version of Pascal MT+86 allows more user space, a full research capability in the long run.

Bibliography

1. Artzy, E. and G.T. Herman. "The Theory, Design, Implementation, and Evaluation of a Three-Dimensional Surface Detection Algorithm," Computer Graphics 14 (3): 2-9 (July 1980).

2. Boinodiris, S. "Hidden Surface Elimination for Complex Graphical Scenes," Computer Graphics 14: 153-167 (March 1981).

3. Brown, C.M. "Fast Display of Well-Tesselated Surfaces," Computers and Graphics 4 (2): 77-85 (1979).

4. Fuch, H., Z.M. Kedem, and B.F. Naylor. "On Visible Surface Generation by a Priori Tree Structures," Computer Graphics 14 (3): 124-133 (July 1980).

5. Goad, Chris. "Special Purpose Automatic Programming for Hidden Surface Elimination," Computer Graphics 16: 167-178 (July 1982).

6. Hamlin, G. and C. Gear. "Raster-Scan Hidden Surface Algorithm Techniques," Interactive Computer Graphics , an IEEE Tutorial from COMPCON 80, San Francisco, 264-271.

7. Hedgley, David R. Jr. "A General Solution to the Hidden-Line Problem", Report from the Ames Research Center, Dryden Flight Research Facility, Edwards California, Nov 1981.

8. Hornung, Christoph. "An Approach to a Calculation -Minimized Hidden Line Algorithm," Eurographics '81, Proceedings of the International Conference and Exhibition, 31-42.

9. Hubschman, H. and S. Zucker. "Frame-to-Frame Coherence and the Hidden Surface Computation: Constraints for a Convex World," Computer Graphics 15 (3): 45-54 (August 1981).

10. Knowlton, K. and L. Cherry. "ATOMS-- A Three-D Opaque Molecular System for Color Pictures of Space-Filling or Ball-and-Stick Models," Computers and Chemistry 1: 161-166 (1977).

11. Laib, G., R. Puk, and G. Stowell. "Integrating Solid Image Capability Into a General Purpose Calligraphic Graphics Package," Computer Graphics 14 (3): 79-85 (July 1980).

12. Myers, A. "An Efficient Algorithm for Computer Generated Pictures," Ohio State University Computer Res. Group, 1975.

13. Newell,M.E., R.G. Newell, and T.L. Sancha "A Solution to the Hidden Surface Problem," Proceedings of the ACM National Conference (1972), 443-450.

14. Reising, John M. and Carol Jean Kopala. "Cockpit Applications of Computer Graphics." Report for Harvard Computer Graphics Week, Harvard University, Graduate School of Design, 1982.

15. Rubin, S. and T. Whitted. "A Three-Dimensional Representation for Fast Rendering of Complex Scenes," Computer Graphics 14 (3): 110-116 (July 1980).

16. "Status Report of the Graphics Standards Planning Committee," Computer Graphics, 13 (3): (August 1979).

17. Stein, Kenneth J. "Technical Survey: Advanced Engineering Simulators, Computer Models Cut USAF Test Costs," Aviation Week & Space Technology, 118 (3): 77-79 (January 17, 1983).

18. Stluka, Federick P. et al. "Overview of the University of Pennsylvania CORE System Standard Graphics Package Implementation," Computer Graphics, 16 (2): 177-86 (June 1982).

19. Sutherland, I.E., and G.W. Hodgman, "Reentrant Polygon Clipping," Communications of the ACM 17 (1): 32-42 (January 1974).

20. Sutherland, I.E., R.F. Sproul, and R.A. Schumacker. "A Characterization of Ten Hidden Surface Algorithms," Computing Surveys 6 (1): 1-55 (March 1974).

21. Watkins, G.S. "A Real Time Visible Surface Algorithm," University of Utah Computer Science Department, UTECH-CSc-70-101 (1970).

Appendix A

Pascal Source Code

```
(*************************************************************************)
(*  Date: 4 Dec 83                                                       *)
(*  Version : 1.0                                                        *)
(*  Name: Resolver                                                       *)
(*  Module Number: 5.2                                                   *)
(*  Function:                                                            *)
(* This procedure is the driver routine for resolving the sorted        *)
(* list made by the sort procedure.  Its algorithm is based on the      *)
(* Newell, Newell, and Sancha method described in " A Solution to       *)
(* the Hidden Surface Problem," Proceedings of the ACM National         *)
(* Conference, Boston, pgs 443-450 (August 1972).  It basically will *)
(* compare a face P against all other faces Q contained in the set    *)
(* of faces that overlap P in z extent.  As soon as we have resolved *)
(* all conflicts either by testing x,y screen extents, the positions *)
(* of P and Q relative to each other using plane equations, the       *)
(* actual screen drawings of all the lines of P and Q, or at last     *)
(* resort splitting P or Q; we scan convert P the furthest face from *)
(* the eyepoint.  By systematically scan converting from back to      *)
(* front, we will create the correct picture.  This method is         *)
(* sometimes referred to as a painter's algorithm because like an     *)
(* artist we will paint and overwrite to create the final picture.    *)
(*  Inputs: None.                                                        *)
(*  Outputs: None.                                                       *)
(*  Global Variables Used: Hiddenlist (hidden surface node pointer)   *)
(*                         The Hidden Surface Linked List Structure   *)
(*  Global Variables Changed: None.                                      *)
(*  Global Tables Used:  None.                                           *)
(*  Global Tables Changed: None.                                         *)
(*  Files Read: None.                                                    *)
(*  Files Written: None.                                                 *)
(*  Modules Called: Polysplit.                                           *)
(*                  Funnel (places primitives on the screen).           *)
(*                  Newframedd (clears the screen).                     *)
(*                  Matrix_Multiply (performs 1X4 * 4X4 matrix mult *)
(*                  Make_Invert_Matrix (creates inverse matrix of  *)
(*                                        image transformation        *)
(*  Calling Modules: End_Batch_of_Updates.                            *)
(*                                                                       *)
(*  Author: Tom S. Wailes, Capt, USAF                                    *)
(*  History: This is the original module                                 *)
(*************************************************************************)

Module RSOLVER;

 const
   {$I defconst.src}
   (*EOC end of const section *)

 type
```

```
  [$I deftype.src]
  (*EOT end of type section *)

var
  [$I extvar.src]
  (*EOV end of var section *)

(*BOP beginning of procedures and functions *)
EXTERNAL [3] PROCEDURE POLY_SPLIT(VAR poly : LINKPTR; VAR splt : LINKPTR;
                                   mde : SPLITTYPE);
EXTERNAL [3] PROCEDURE NEWFRMDD;
EXTERNAL [3] PROCEDURE FUNNEL(pt : PRIM_PTR);
EXTERNAL   PROCEDURE MAKEINVAT(TFORM:MATRIX;VAR INVMAT:RARRAY4X4);
EXTERNAL   PROCEDURE MATRIX_MUL(A: VECTOR4; B: RARRAY4X4; C: VECTOR4);
(*EOP end of procedures and functions *)



procedure RESOLVER;

const
 eps = 0.000000001;        (* close enough to zero for testing *)
                           (* needed because of computer round off *)
type
 position = (behind,in_front);

var
 swap,split_p, split_q : boolean;
 p_face,q_face,I,psides,qsides : integer;
 P,Q : LINKPTR;
 scren_x,scren_y : IARRAY;
 transmat : RARRAY4x4;
 vec : VECTOR4;
 p_values,q_values : RARRAY;
```

```
(*********************************************************************)
function plane_test(sides: INTEGER;VAR face: INTEGER;VAR value: RARRAY;
          A : LINKPTR; SIGN : POSITION; B : LINKPTR): BOOLEAN ;

(**********************************************************************)
(*  Date: 4 Dec 83                                                     *)
(*  Version : 1.0                                                      *)
(*  Name: Plane_Test                                                  *)
(*  Function:                                                          *)
(* This function tests the relative position of a primitive by substi-  *)
(* tuting the points of A into the plane equations of B. If all points  *)
(* are on the side of the sign then true is returned, else a false value*)
(* will be returned, with the number of sides compared in face. The user*)
(* sets face prior to the call to prevent uneeded multiplications when  *)
(* performing the reversal tests in the last checks before spliting     *)
(*  Inputs:  sides (number of sides of the polygon)                    *)
(*           face (number of sides already tested)                     *)
(*           A ( pointer to the polygon)                               *)
(*           sign (flag that determines if testing in front or behind  *)
(*                the plane )                                          *)
(*           B (pointer to plane)                                      *)
(*  Outputs: face (number of sides tested)                             *)
(*           Value (array of plane values calculated)                  *)
(*           Boolean value of whether the polygon was on the right side *)
(*           of the plane                                              *)
(*  Global Variables Used:  Eps (allowance for roundoff)              *)
(*  Global Variables Changed:    None.                                 *)
(*  Global Tables Used: None.                                          *)
(*  Global Tables Changed: None.                                       *)
(*  Files Read:  None.                                                 *)
(*  Files Written:   None.                                             *)
(*  Modules Called:  None.                                             *)
(*  Calling Modules: Resolver.                                         *)
(*                                                                     *)
(*  Author: Tom S. Wailes, Capt, USAF                                  *)
(*  History: This is the original module                               *)
(**********************************************************************)


begin
case sign of
 in_front :
   begin
     repeat
        face :=  face + 1;
               (* plane equation is Ax + By + Cz + D = 0               *)
               (* all points that satisfy this equation are in the plane *)
               (* that has the the coefficients A,B,C, and D            *)
               (* if Ax + By + Cz + D <> 0 then the sign of the value   *)
               (* tells us which side of the plane the point is located  *)
               (* if a point in A when substituted into the plane of B   *)
               (* results in a positive value, then the point is in front *)
        value[face]  := ( B^.plane_a * A^.hidden_x_coor[face]) +
                              ( B^.plane_b * A^.hidden_y_coor[face]) +
                              ( B^.plane_c * A^.hidden_z_coor[face]) +
                                B^.plane_d;
      until ((value[face] + eps < 0) or (face >= sides));

       PLANE_TEST := value[face] + eps >= 0;  (* set return value *)
    end;
  behind :
    begin
        repeat
         face := face + 1;
              (* substitute the coordinates of A into the plane of B *)
              (* if the plane value is always negative then A is behind *)
          value[face] := ( B^.plane_a * A^.hidden_x_coor[face]) +
                              ( B^.plane_b * A^.hidden_y_coor[face]) +
                              ( B^.plane_c * A^.hidden_z_coor[face]) +
                                B^.plane_d;
       until ((value[face] - eps > 0) or (  face >=  sides));

       PLANE_TEST := value[face] - eps <= 0;   (* set return values *)
    end;  (* behind *)
   end; (* case *)
 end; (* of plane_test *)
```

```
function SCREEN_OVERLAP( P : linkptr; sides_p : integer;
                         Q : linkptr; sides_q : integer): boolean;
(***************************************************************************)
(*  Date: 4 Dec 83                                                       *)
(*  Version : 1.0                                                        *)
(*  Name: Screen_Overlap.                                               *)
(*  Function:                                                            *)
(* This function tests for the actual overlap on the screen of the lines*)
(* comprising the edges of the polygons.  This is done by calculating   *)
(* the parametric line equation constant T for the intersection of each *)
(* P edge and Q edge.  If the T value for both the P and Q edges are    *)
(* between 0 and 1 then an overlap has occured and the polygons will    *)
(* have to be split.  Additionally if a edge of P when extended in both *)
(* directions splits only an odd number of the sides of Q then overlap  *)
(* will also occur.                                                      *)
(*  Inputs:    P, Q (pointers to the polygons to be checked)            *)
(*             sidesp,sidesq (number of sides to the polygons)          *)
(*  Outputs:  Boolean value of whether there is screen overlap          *)
(*  Global Variables Used: View_State_projection type (parallel or    *)
(*                                                     perspective)    *)
(*  Global Variables Changed: None.                                     *)
(*  Global Tables Used: None.                                           *)
(*  Global Tables Changed: None.                                        *)
(*  Files Read:   None.                                                 *)
(*  Files Written:  None.                                               *)
(*  Modules Called:  None.                                              *)
(*  Calling Modules: Resolver.                                          *)
(*                                                                       *)
(*  Author: Tom S. Wailes, Capt, USAF                                   *)
(*  History: This is the original module                                *)
(***************************************************************************)


var
 overlap : boolean;
 i,p1,p2,q1,q2,Q_count : integer;
 P_count : IARRAY;
 Q_x_coor,Q_y_coor : RARRAY;
 p_delta_x,p_delta_y,T,Tprime : real;
 P_x1,P_x2,P_y1,P_y2,Q_x1,Q_x2,Q_y1,Q_y2 : real;


begin
      p1 := 1;
      p2 := 2;
      for i := 1 to sides_q do   (* initialize the number of times that *)
           P_count[i] := 0;      (* Q crossed the edges of P.           *)

      (* Save multiple calculations of the P and Q screen coordinates if *)
      (* perspective projection is requested *)

     if view_state.pjtyp = 2 then (*perspective projection *)
        begin
        P_x2 := P^.hidden_x_coor[1]/P^.hidden_w_coor[1];  (* init first *)
        P_y2 := P^.hidden_y_coor[1]/P^.hidden_w_coor[1];  (* P edge val *)

        for i:= 1 to sides_q do     (* calculate all Q edge values *)
          begin
            Q_x_coor[i] := Q^.hidden_x_coor[i]/Q^.hidden_w_coor[i];
            Q_y_coor[i] := Q^.hidden_y_coor[i]/Q^.hidden_w_coor[i];
          end;
      end
    else          (* parallel case *)
      begin
       P_x2 := P^.hidden_x_coor[1];   (* init first P edge value *)
       P_y2 := P^.hidden_y_coor[1];

        for i:= 1 to sides_q do   (* store all Q edge val for parallel *)
          begin
            Q_x_coor[i] := Q^.hidden_x_coor[i];
            Q_y_coor[i] := Q^.hidden_y_coor[i];
          end;
      end;
    overlap := false;
    while (( not overlap) and ( p1 <= sides_p )) do  (* test all faces *)
     begin                                          (* in P          *)
      q1 := 1;          (* initialize the q pointers for each P face *)
      q2 := 2;
      Q_count := 0; (* init the counter  that determine how many  *)
                    (* times a P side intersects the a Q polygon edges*)
                    (* If ever a side intersects an odd # of times *)
                    (* overlap has certainly occured *)

      P_x1 := P_x2;  (* set the P edge beginning point to the last *)
      P_y1 := P_y2;  (* point coordinate checked *)

                     (* calculate the ending point of the edge if needed *)

    if view_state.pjtyp = 2 then (*perspective projection *)
      begin
        P_x2 := P^.hidden_x_coor[p2]/P^.hidden_w_coor[p2];
        P_y2 := P^.hidden_y_coor[p2]/P^.hidden_w_coor[p2];
      end
    else          (* parallel case *)
      begin
       P_x2 := P^.hidden_x_coor[p2];
       P_y2 := P^.hidden_y_coor[p2];
      end;

               (* save duplication of computation of    *)
               (* the change in x and y for each P face *)

      p_delta_x := P_x2 - P_x1;
      p_delta_y := P_y2 - P_y1;

      while (( not overlap) and ( q1 <= sides_q)) do  (* for each P face *)
       begin
```

```
          Q_xl := Q_x_coor[ql];
          Q_x2 := Q_x_coor[q2];
          Q_yl := Q_y_coor[ql];
          Q_y2 := Q_y_coor[q2];

Tprime := ((( P_xl - Q_xl) * p_delta_y)
           + (( Q_yl - P_yl) * p_delta_x))
           / ((( Q_x2 - Q_xl) * p_delta_y)
           - (( Q_y2 - Q_yl) * p_delta_x));

          T := ( Q_xl - P_xl
             + (Tprime * ( Q_x2 - Q_xl)))
             / p_delta_x;

if (( Tprime < 1) and (Tprime >0)) then
 if (( T < 1) and ( T > 0)) then (* overlap has occured *)
     overlap := true
  else
       if T > 1 then
          Q_count := Q_count + 1;     (* the line of pl-->p2 when sent to *)
                                    (* infinity crossed this Q polygon edge*)
  if ((Tprime > 1) and (T < 1) and (T > 0)) then  (* the line of ql-->q2 *)
          P_count[ql] := P_count[ql] + 1;          (* crossed this P edge *)

            ql := ql + 1;
            q2 := q2 + 1;
            if ql = sides_q then q2 := 1; (* last time thru connect last*)
                                          (* vertex to original *)
          end;     (* all Q sides tested against this P *)

       if (Q_count MOD 2 > 0) then   (* if the line of pl-->p2 has *)
          overlap := true;              (* crossed the Q polygon an odd *)
                                        (* number of times, then overlap *)
                                        (* has occured *)

     pl := pl + 1;
     p2 := p2 + 1;
     if pl = sides_p then p2 := 1;  (* last vertex is drawn to beginning *)
  end;
 i := 1;
 while ((not overlap) and (i <= qsides)) do  (* determine if a face in Q *)
    begin
      if (P_count[i] MOD 2 > 0) then        (* has crossed P an odd number *)
        overlap := true;                    (* of times *)
      i := i + 1;
    end;

SCREEN_OVERLAP := overlap;

end; (* overlap function *)
```

```
function on_both_sides(sides,face : INTEGER; values : RARRAY;
                       A,B : LINKPTR): BOOLEAN;
(************************************************************************)
(*  Date: 4 Dec 83                                                      *)
(*  Version : 1.0                                                       *)
(*  Name: On_Both_Sides                                                *)
(*  Function:                                                           *)
(* This function determines if points in the polygon lie on both sides *)
(* of the plane.  It takes advantage of previous calulations saved in  *)
(* the value array, and only does the plane value calculations if needed*)
(*  Inputs:   sides ( the number of sides in the polygon)               *)
(*            face ( the number of sides already tested)                *)
(*            values ( array of calculated plane values )               *)
(*            A ( pointer to the polygon)                               *)
(*            B ( pointer to the plane)                                 *)
(*  Outputs: Boolean value of whether the polygon has points on both   *)
(*                                                              sides  *)
(*  Global Variables Used: Eps (allowance for round off)                *)
(*  Global Variables Changed:  None.                                    *)
(*  Global Tables Used: None.                                           *)
(*  Global Tables Changed:  None.                                       *)
(*  Files Read:   None.                                                 *)
(*  Files Written:  None.                                               *)
(*  Modules Called:  None.                                              *)
(*  Calling Modules: Resolver.                                          *)
(*                                                                      *)
(*  Author: Tom S. Wailes, Capt, USAF                                   *)
(*  History: This is the original module                                *)
(************************************************************************)


var
count : integer;
positive, negative : boolean;
plane_value : real;

begin
 count := 0;
 positive := false;
 negative := false;

 repeat
  count := count + 1;
  if values[count] < -eps then    (* first check calculated values *)
     negative := true          (* to see if points lie on both sides *)
  else if values[count] > eps then
     positive := true;
 until ((negative and positive) or (count >= face));

 if (negative and positive) then  (* exit points on both sides *)
    on_both_sides := true
 else                            (* check remaining points *)
   begin                         (* if any points remain to *)
```

```
    if count < sides then         (* be calculated            *)
     repeat
      count := count + 1;
      plane_value := ( B^.plane_a * A^.hidden_x_coor[count]) +
                     ( B^.plane_b * A^.hidden_y_coor[count]) +
                     ( B^.plane_c * A^.hidden_z_coor[count]) +
                       B^.plane_d;
      if plane_value < -eps then    (* check new calculated values *)
        negative := true          (* to see if points lie on both sides *)
      else if plane_value > eps then
        positive := true;
    until ((negative and positive) or (count >= sides));

    on_both_sides := negative and positive;
  end;
end;
```

```
(***********************************************************************)
(*            main resolver procedure                                 *)
(***********************************************************************)


begin

newfrmdd;  (* clear the screen for drawing *)


P := hiddenlist;   (* point the P pointer to the furthest primitive *)
Q := P^.forwrd;   (* initialize Q *)

while P <> nil do    (* exhaust the list of primitives *)


  begin
    while (( Q^.z_max_view > P^.z_min_view) and ( Q <> nil)
         and (P^.output_primitive^.prims = pgon)) do
                                 (* examine all faces that overlap P *)
                                 (* in z extent                      *)
      begin
      (* test for overlap of P and Q in x,y screen extents *)

      if (( P^.x_max_screen <= Q^.x_min_screen) or   (* if any of these *)
          ( Q^.x_max_screen <= P^.x_min_screen) or   (* conditions exist *)
          ( P^.y_max_screen <= Q^.y_min_screen) or   (* the two polygons *)
          ( Q^.y_max_screen <= P^.y_min_screen))     (* cannot overlap *)
       then
        Q := Q^.forwrd         (* update Q and exit the tests *)
     else
      begin      (* x,y screen extent failure, test plane equations *)
      psides := P^.output_primitive^.coorcnt; (* initialize for further *)
      qsides := Q^.output_primitive^.coorcnt; (* tests                *)
      split_p := false;
      split_q := false;

(* test first for P on the back side of Q   *)
      p_face := 0;
      if plane_test(p_sides,p_face,p_values,P,behind,Q) then
                                             (* all of P is behind Q *)
       Q := Q^.forwrd                        (* move Q and exit tests *)
     else
      begin    (* try again; test now for Q totally in front of P *)


      q_face := 0;
      if plane_test(q_sides,q_face,q_values,Q,in_front,P) then
                                             (* all of Q in front P *)
                                             (* move Q and exit tests *)
        Q := Q^.forwrd
     else
      begin    (* normal plane tests have failed therefore now try*)
               (* projecting the screen points and determining if *)
```

```
                (* the edges actually overlap on the screen surface *)


  if not screen_overlap(P,psides,Q,qsides) then (* polygons do not overlap *)
     Q := Q^.forwrd
  else

(* The last possible alternative before polygon splitting is to reorder *)
(* P and Q in the hope that by simply moving Q in the list we can resolve *)
(* the conflict.  Since the extent and overlap tests have failed all we  *)
(* need to check is the reverse ordering of the plane equation tests. *)

  begin
    if Q^.moved then   (* reordering is only allowed once to prevent *)
      begin              (* cyclic overlap of polygons               *)
        if on_both_sides(p_sides,p_face,p_values,P,Q) then
           begin
            poly_split(P,Q,Q_split_P);  (* points in P on both sides of Q *)
           end
         else if on_both_sides(q_sides,q_face,q_values,Q,P) then
          begin
            poly_split(Q,P,P_split_Q);  (* points in Q on both sides of P *)
          end
        else
          BEGIN
            Q^.moved := true;
            Q^.backwrd^.forwrd := Q^.forwrd;  (* remove Q *)
            if Q^.forwrd <> nil then
               Q^.forwrd^.backwrd := Q^.backwrd;
            if P^.backwrd <> nil then           (* insert Q "behind" P *)
            P^.backwrd^.forwrd := Q;
            Q^.forwrd := P;           (* link Q pointing to P *)
            Q^.backwrd := P^.backwrd;
            P^.backwrd := Q;
            P := Q;
            Q := P^.forwrd^.forwrd; (* skip old P, its resolved *)

          END;
      end
    else
      begin
(* We can use the results of previous calculations in this test.  At this *)
(* point, we know that there was at least one vertex found to be on the   *)
(* observer's side of Q.  If all points of P when substituted into the    *)
(* plane equation of Q are found to be on the observer's side, then Q can *)
(* and should be scan converted before P.  This involves making the present *)
(* Q become P and moving the present Q "behind" P in the linked list of    *)
(* primitives.  Furthermore, we can easily test for the possibility of     *)
(* verticies of P on both sides of Q.  If split_p is not set then the first *)
(* vertex of P was on the observers side of Q.  Thus we will first test    *)
(* split_p and if not set check the rest of the verticies to see if they   *)
(* also are on the observer's side.  Otherwise, we will use the same type  *)
(* of reasoning to see if Q is totally on the back side of P.              *)
```

```
  swap := false;
  split_p := on_both_sides(p_sides,p_face,p_values,P,Q);
  if not split_p then
     swap := true        (* move Q behind P in list, *)
  else
     begin
        split_q := on_both_sides(q_sides,q_face,q_values,Q,P);
        if not split_q then
           swap := true;          (* move Q behind P in list, *)
    end;
   if swap then
    begin                         (* mark Q as moved, make it the new P *)
     Q^.moved := true;
     Q^.backwrd^.forwrd := Q^.forwrd;  (* remove Q *)
     if Q^.forwrd <> nil then
        Q^.forwrd^.backwrd := Q^.backwrd;
     if P^.backwrd <> nil then             (* insert Q in "back" of P *)
        P^.backwrd^.forwrd := Q;
     Q^.forwrd := P;                       (* link Q pointing to P *)
     Q^.backwrd := P^.backwrd;
     P^.backwrd := Q;
     P := Q;
     Q := P^.forwrd^.forwrd;  (* skip old P because we know its resolved *)

    end
else            (* time for splitting polygons *)
 begin
  if split_p then
     poly_split(P,Q,Q_split_P)   (* points in P on both sides of Q *)
  else if split_q then
     poly_split(Q,P,P_split_Q)   (* points in Q on both sides of P *)
        else
          writeln('** resolver: swap not made points UNRESOLVED');
 end;

    end; (* else for Q^.moved *)
   end; (* else overlap *)
  end; (* Q in front of P *)
 end; (* P behind Q *)
end; (* screen overlaps *)
end; (* while Q^.zmax and <> nil *)

if (( Q = nil) or ( not Q^.moved)) then
  begin
    funnel(P^.output_primitive);  (* scan convert P *)

(* if the values of the coordinates have changed then we must recalculate *)
(* the NDC equivalent before leaving.  This only applies to the "split" and *)
(* "new_prim" output primitives we have made this time                    *)

    with P^ do
       begin
        if (( enter = split) or ( enter = new_prim)) then
          begin      (* create a new inverse transformation matrix *)
```

```
                makeinvat(output_primitive^.owner_seg^.image_trans,transmat);
                for I := 1 to output_primitive^.coorcnt do  (* matrix mult *)
                 begin
                  vec[1] := hidden_x_coor[i];
                  vec[2] := hidden_y_coor[i];
                  vec[3] := hidden_z_coor[i];
                  vec[4] := hidden_w_coor[i];
                  matrix_mul(vec,transmat,vec);
                  output_primitive^.xcoor[i] := vec[1];
                  output_primitive^.ycoor[i] := vec[2];
                  output_primitive^.zcoor[i] := vec[3];
                  output_primitive^.wcoor[i] := vec[4];
                 end;
             end;
          end;
       P := P^.forwrd;
        if P <> nil then
           Q := P^.forwrd;
     end
   else
    Q := Q^.forwrd;
   end; (* P <> nil *)
   end;
   modend.
```

```
(***********************************************************************)
(*  Date: 4 Dec 83                                                      *)
(*  Version : 1.0                                                       *)
(*  Name: Poly_Split                                                   *)
(*  Module Number: 5.2.2                                                *)
(*  Function:                                                           *)
(* This module  takes a polygon and splitting  plane  and  divides     *)
(* the polygon into two distinct polygons separated by the splitting *)
(* plane.  In addition, the resulting fragments are placed back into *)
(* the hidden surface linked list structure for future processing.   *)
(* The main procedure will actually split the polygon and the smaller*)
(* placement procedure will place the fragments into the linked list.*)
(*  Inputs:     Polygon ( a pointer to polygon to be split)            *)
(*              Splitpln ( a pointer to the splitting plane )          *)
(*              mode (flag that specifies if P is split by Q, or if  *)
(*                     Q is split by P)                                *)
(*  Outputs:    Polygon ( a pointer to the moved polygon position)    *)
(*              Splitpln ( a pointer to the moved splitting plane)    *)
(*  Global Variables Used: The primitive and Hidden Surface data       *)
(*                                   Structures                         *)
(*  Global Variables Changed: The Hidden Surface data Structure        *)
(*  Global Tables Used:  None.                                           *)
(*  Global Tables Changed: None.                                        *)
(*  Files Read:  None.                                                  *)
(*  Files Written: None.                                                *)
(*  Modules Called:  Initialize Hidden Surface Node                     *)
(*  Calling Modules: Resolver.                                          *)
(*                                                                      *)
(*  Author: Tom S. Wailes, Capt, USAF                                   *)
(*  History: This is the original module                                *)
(***********************************************************************)


Module SPLITER;

 const
   [$I defconst.src]
   eps = 0.000000001;    (* close enough to zero for comparisons,needed *)
                         (* because of computer round off *)
(*EOC end of const section *)

 type
   [$I deftype.src]
   (*EOT end of type section *)

 var
   [$I extvar.src]
   (*EOV end of var section *)

 (*BOP beginning of procedures and functions *)
EXTERNAL [3] PROCEDURE INIT_HIDDEN_LINK(PT : PRIM_PTR; MDE : ENTRY);

 (*EOP end of procedures and functions *)
```

```
procedure POLY_SPLIT( VAR polygon : LINKPTR; VAR splitpln : LINKPTR;
                      mode : SPLITTYPE);

var
pl,p2,side_a,side_b : integer;
plane_pl,plane_p2,delta_x,delta_y,delta_z,delta_w,T_intersection,
x_intersection,y_intersection,z_intersection,w_intersection : real;
x_a,y_a,z_a,w_a,x_b,y_b,z_b,w_b : RARRAY;
first_plane_pl : real;
```

```
procedure place_node(start,new_hidden : LINKPTR);

(*********************************************************************)
(*  Date: 4 Dec 83                                                   *)
(*  Version : 1.0                                                    *)
(*  Name: Place Node                                                 *)
(*  Function:                                                        *)
(*   This procedure used to properly place a hidden node created     *)
(*   after a polygon split.  It starts at the splitting plane and    *)
(*   searches toward the observer until it finds its proper place    *)
(*   in the list using its z extent for the test.  It will ignore    *)
(*   polygons that have been moved when testing.  Finally it does    *)
(*   the actual insertion into the list.                             *)
(*  Inputs:  Start ( beginning point in hidden surface linked list   *)
(*           at which to start searching for proper z placement )    *)
(*           New_Hidden (pointer to node to be inserted)            *)
(*  Outputs: None.                                                   *)
(*  Global Variables Used: Hidden surface linked list structure      *)
(*  Global Variables Changed:  Hidden surface linked list structure  *)
(*  Global Tables Used:  None.                                       *)
(*  Global Tables Changed: None.                                     *)
(*  Files Read:   None.                                              *)
(*  Files Written:  None.                                            *)
(*  Modules Called:  None.                                           *)
(*  Calling Modules: Placement.                                      *)
(*                                                                   *)
(*  Author: Tom S. Wailes, Capt, USAF                                *)
(*  History: This is the original module                             *)
(*********************************************************************)

var
 present : LINKPTR;

begin
     present := start^.forwrd;  (* set temporary pointer *)
     if present = nil then
       begin
         start^.forwrd := new_hidden; (*place on right *)
         new_hidden^.backwrd := start; (* end of hidden *)
         new_hidden^.forwrd := nil;      (* linked list *)
       end
     else
       begin
        (* The next lines of logic testing will ensure that primitives *)
        (* are ordered first by their z maximum values. If primitives *)
        (* share the same z maximum value the sort will result in the *)
        (* primitive with the smallest minimum value being closest to *)
        (* the eye. Moved primitives cannot be trusted for their z values *)
        (* and therefore will be ignored. I also dont want to go past  *)
        (* the last primitive in the linked list.                      *)

        while (((new_hidden^.z_max_view < present^.z_max_view) or
               ((new_hidden^.z_max_view = present^.z_max_view) and
                  (new_hidden^.z_min_view < present^.z_min_view)) or
                   present^.moved) and (present^.forwrd <> nil)) do
begin

                          present := present^.forwrd;
end;

          if ((new_hidden^.z_max_view < present^.z_max_view) or
                ((new_hidden^.z_max_view = present^.z_max_view) and
                 (new_hidden^.z_min_view < present^.z_min_view)) or
                  present^.moved) then    (* I hit the end of the list *)
                   begin
                     present^.forwrd := new_hidden; (* place on right *)
                     new_hidden^.backwrd := present; (* end of hidden *)
                     new_hidden^.forwrd := nil;      (* linked list *)
                   end
               else
                 begin          (* normal insert *)
                 new_hidden^.forwrd := present;
                   new_hidden^.backwrd := present^.backwrd;
                   if present^.backwrd <> nil then
                      present^.backwrd^.forwrd := new_hidden;
                   present^.backwrd:= new_hidden;
                 end;
        end; (* test for nil *)
end;  (* of  place_node *)
```

```
procedure placement;

(*********************************************************************)
(*  Date: 4 Dec 83                                                   *)
(*  Version : 1.0                                                    *)
(*  Name: Placement                                                  *)
(*  Function:                                                        *)
(* This sub_procedure needed to break the poly split procedure into *)
(* two distinct parts.  The MT+86 compiler has a limited space      *)
(* allotted for procedure area.  This encourages modularity and is  *)
(* really for the best.  This procedure will take the two vertex    *)
(* lists made during the main splitting routine and the splitting   *)
(* polygon and merge them into the correct ordering into the hidden *)
(* surface linked list structure.                                   *)
(*  Inputs: None.                                                    *)
(*  Outputs: None.                                                   *)
(*  Global Variables Used:  vertex lists for sides a and b, Mode    *)
(*  Global Variables Changed: Hidden surface linked list structure  *)
(*                            Primitive data structure              *)
(*  Global Tables Used: None.                                        *)
(*  Global Tables Changed: None.                                     *)
(*  Files Read:  None.                                               *)
(*  Files Written: None.                                             *)
(*  Modules Called:  Place node, Initialize hidden surface node     *)
(*  Calling Modules: Polygon_Split                                   *)
(*                                                                   *)
(*  Author: Tom S. Wailes, Capt, USAF                                *)
(*  History: This is the original module                             *)
(*********************************************************************)


var
  I : integer;
  B,orig_poly : PRIM_PTR;
  last,temp_p,Bptr : LINKPTR;


begin


(* place negative verticies from side_b into original primitive *)
 polygon^.output_primitive^.coorcnt := side_b;  (* set # of points *)
 for i := 1 to side_b do       (* move points to linked list area *)
   begin
    polygon^.hidden_x_coor[i] := x_b[i];
    polygon^.hidden_y_coor[i] := y_b[i];
    polygon^.hidden_z_coor[i] := z_b[i];
    polygon^.hidden_w_coor[i] := w_b[i];
(* polygon^.output_primitive^.red_index_verticies[i] := red_b[i];  *)
  end;

(* reinitialize the extents, plane equations, etc for this modified entry *)
```

```
 init_hidden_link(polygon^.output_primitive,split);

(* create new primitive data holder and initialize the hidden node for  *)
(* the half of the polygon on the positive side of the split plane *)

 new(B,pgon);
 origpoly := polygon^.output_primitive;
   with B^ do
     begin
       prims := origpoly^.prims;
       owner_seg := origpoly^.owner_seg;
       next_prim := origpoly^.next_prim;
       origpoly^.next_prim := B;          (* place B in list *)
       pick_id := origpoly^.pick_id;
       penval :=  origpoly^.penval;
       line_index := origpoly^.line_index;
       poly_interior_style := origpoly^.poly_interior_style;
       poly_edge_style := origpoly^.poly_edge_style;
       fill_index := origpoly^.fill_index;
       coorcnt := side_a;
       new(Bptr);
       hidden := Bptr;
       for i := 1 to side_a do  (* transfer the point values *)
         with Bptr^ do
           begin
            hidden_x_coor[i] := x_a[i];
            hidden_y_coor[i] := y_a[i];
            hidden_z_coor[i] := z_a[i];
            hidden_w_coor[i] := w_a[i];
         (* B^.red_index_verticies[i] := red_a[i];  *)
           end;
(* initialize the extents, plane equations, etc for this new entry *)
  init_hidden_link(B,new_prim);

   end;

case mode of
     q_split_p: begin
                 (* P was split so leave negative part in place *)
                 (* and place new positive part in front of Q *)

                 place_node(splitpln,Bptr);
                 Bptr^.moved := true;          (* z may be out of bounds *)
                 splitpln := splitpln^.forwrd; (* move Q to check next *)
               end;                            (* primitive in list *)
                                               (* P does not change *)
     p_split_q:  begin
                 (* Q was split so place negative part behind P and *)
                 (* place new (positive) part in front of P *)
         place_node(splitpln,Bptr); (* positive part in front of P*)

         polygon^.backwrd^.forwrd := polygon^.forwrd; (* remove Q *)
         if polygon^.forwrd <> nil then
            polygon^.forwrd^.backwrd := polygon^.backwrd;
```

```
                                          (* place negative polygon *)
          if splitpln^.backwrd <> nil then     (* on back side of P, *)
             splitpln^.backwrd^.forwrd := polygon;  (* further from eye *)
          polygon^.forwrd := splitpln;
          polygon^.backwrd := splitpln^.backwrd;
          splitpln^.backwrd := polygon;
          polygon^.moved := true;
          splitpln := polygon;    (* make this negative portion the new P *)
          polygon := splitpln^.forwrd^.forwrd;  (* move Q to next prim after*)
         end;                                   (* old P *)

 end; (* case *)
end;    (* placement *)
```

```
(***************************************************************************)
(*                                                                         *)
(* This starts the main polygon  splitting procedure needed to split       *)
(* conflicting polygons found by the resolver subroutine.  It uses the     *)
(* plane equation of the input "split" to divide a "polygon" into two      *)
(* polygons. The main algorithm is taken from the article found in :       *)
(* Sutherland, I.E. and G.W. Hodgman. " Reentrant Polygon Clipping,"       *)
(* Communications of the ACM, 17(1) : 32-42 (January 1974).  Basically     *)
(* we take each vertex in turn and see which side of the plane that        *)
(* it resides.  If it sits on the opposite side of the plane from the      *)
(* last vertex we checked then we must calculate the intersection of the *)
(* line between the verticies and the splitting plane.  This new vertex    *)
(* is then included in the list of verticies for both sides.  As we        *)
(* continue around the polygon, points on the positive side of the plane *)
(* are kept in one list, and points on the negative side are kept in       *)
(* another.  At the end, we end up with two sets of verticies that         *)
(* represent the two new polygons.  We then call placement to create an   *)
(* initialized node and place the  resulting pieces back into the  hidden*)
(* surface linked list.                                                    *)
(*                                                                         *)
(***************************************************************************)


begin

(* determine the placement of the first point *)
 side_a :=0;
 side_b := 0;  (* initialize the number of verticies in each list *)

plane_pl := 0.0; (* initialize the first value to fall through test for *)
                 (* intersection                                        *)

pl := 0;     (* set the pointers so that p2 will initially be pointing to *)
p2 := 1;     (* the first vertex in the list *)

(* process main body of vertex points *)

while p2 <= polygon^.output_primitive^.coorcnt do
 begin
   (* determine the plane values of points Pl and P2, the verticies *)
   (* in question.  Plane_pl already set from previous calculation  *)
   (* either from initial conditions  or previous time through loop.*)

plane_p2 := (splitpln^.plane_a * polygon^.hidden_x_coor[p2])
          + (splitpln^.plane_b * polygon^.hidden_y_coor[p2])
          + (splitpln^.plane_c * polygon^.hidden_z_coor[p2])
          +  splitpln^.plane_d ;
if p2=1 then      (* save initial value  *)
 first_plane_pl := plane_p2;

  (* Does the new point reside on the splitting plane?   *)
  (* if it does put it in both vertex lists and continue *)
  (* otherwise we must check to see if the plane was cut *)
```

```
if (( plane_p2 < 0.0 + eps) and ( plane_p2 > 0.0 - eps)) then (* on plane *)
    with polygon^ do
     begin
       side_a := side_a + 1;  (* place the vertex in both lists *)
       side_b := side_b + 1;
       x_a[side_a] := hidden_x_coor[p2];
       y_a[side_a] := hidden_y_coor[p2];
       z_a[side_a] := hidden_z_coor[p2];
       w_a[side_a] := hidden_w_coor[p2];
       x_b[side_b] := hidden_x_coor[p2];
       y_b[side_b] := hidden_y_coor[p2];
       z_b[side_b] := hidden_z_coor[p2];
       w_b[side_b] := hidden_w_coor[p2];
       plane_p2 := 0.0;  (* set to zero to prevent intersection tests *)


(* because the CORE supports color intensities at each vertex the color *)
(* values would also have to moved from the cooresponding vertex into   *)
(* a temporary area similar to the x,y,z values shown above!!!!         *)

    end
  else
    begin

(* Does the line P1 --> P2 cross the plane?  If it does then the signs *)
(* of plane_p1 and plane_p2 will be different  *)

     if ((( plane_p1 < 0) and ( plane_p2 > 0)) or
        (( plane_p1 > 0) and ( plane_p2 < 0))) then
       begin
         with polygon^ do                   (* calculate intersection using *)
          begin                             (* parametric equations of lines *)
           delta_x := hidden_x_coor[p2]     (* solving for T; for we know    *)
                    - hidden_x_coor[p1];    (* intersection must be in the   *)
           delta_y := hidden_y_coor[p2]     (* plane and at the same time on *)
                    - hidden_y_coor[p1];    (* the line P1--> P2. Since these*)
           delta_z := hidden_z_coor[p2]     (* values will be used more than *)
                    - hidden_z_coor[p1];    (* once, do them here *)
           delta_w := hidden_w_coor[p2]
                    - hidden_w_coor[p1];
          end;

(* calculate the parametric line equation constant T for the intersection *)
(* of the line P1 -->P2 and the split plane *)
        with splitpln^ do
          T_intersection := (-plane_p1)/((plane_a * delta_x) +
                                         (plane_b * delta_y) +
                                         (plane_c * delta_z));


        (* now calculate the x,y,z of the intersection point by *)
        (* substituting T into the parametric line equations for x,y,z *)

        with polygon^ do
```

```
              begin
               x_intersection := hidden_x_coor[pl] + (T_intersection * delta_x);
               y_intersection := hidden_y_coor[pl] + (T_intersection * delta_y);
               z_intersection := hidden_z_coor[pl] + (T_intersection * delta_z);
               w_intersection := hidden_w_coor[pl] + (T_intersection * delta_w);


(* A similar calculation can be performed to find the color values needed *)
(* this point.  Using T and knowing the values of say RED at Pl and P2 we *)
(* can calculate RED at the intersection vertex by :                      *)
(*                 RED[pl] := RED[pl] + (T_intersection * (delta_red));    *)
(* This new value could then be placed into the vertex list for the new   *)
(* point                                                                  *)

             end;
              (* place the intersection point in both vertex lists *)
             side_a := side_a + 1;
             side_b := side_b + 1;
             x_a[side_a] := x_intersection; x_b[side_b] := x_intersection;
             y_a[side_a] := y_intersection; y_b[side_b] := y_intersection;
             z_a[side_a] := z_intersection; z_b[side_b] := z_intersection;
             w_a[side_a] := w_intersection; w_b[side_b] := w_intersection;
      end;

(* now place the p2 point in the proper list *)

with polygon^ do
begin
 if plane_p2 < 0 then  (* point on negative side of plane *)
    begin
      side_b := side_b + 1;
      x_b[side_b] := hidden_x_coor[p2];
      y_b[side_b] := hidden_y_coor[p2];
      z_b[side_b] := hidden_z_coor[p2];
      w_b[side_b] := hidden_w_coor[p2];
(* here you would move the color into side b's color vertex array *)
    end
 else     (* the point is on the positive side of the plane *)
   begin
        side_a := side_a + 1;
        x_a[side_a] := hidden_x_coor[p2];
        y_a[side_a] := hidden_y_coor[p2];
        z_a[side_a] := hidden_z_coor[p2];
        w_a[side_a] := hidden_w_coor[p2];
        (* color verticies for side_a moved here *)
   end;
 end; (* with polygon^ *)
end; (* non zero plane_p2 *)

pl := pl + 1;
p2 := p2 + 1;
plane_pl := plane_p2;  (* saving this value prevents duplicate calculations
*)
```

```
end;  (* while for main vertex points *)

(* at this point all we need to do is close the vertex lists. If the line  *)
(* from the last vertex to the first vertex crosses the plane the          *)
(* intersection point must be sent to each list.  Plane_pl contains the    *)
(* substitution into the plane equation for the last point and of course   *)
(* first_plane_pl contains the value from the initial vertex.  In the      *)
(* initial algorithm a test was made to see if there was output from the   *)
(* split.  This was because their algorithm was used to split polygons on  *)
(* screen surfaces and polygons completely contained on 1 surface would not *)
(* be split.  Because I will have a split every time, I will instead test   *)
(* for degenerate polygons when I terminate *)


 p2 := 1;                      (* set p2 to point to the first vertex *)
 plane_p2 := first_plane_pl;
 if ((( plane_pl < 0.0 - eps) or ( plane_pl > 0.0 + eps)) and
     (( plane_p2 < 0.0 - eps) or ( plane_p2 > 0.0 + eps))) then

 (* if either the beginning point or ending point is on the plane then *)
 (* it is already included in the vertex lists!!!!                     *)


 if (((plane_pl < 0) and (plane_p2 >0)) or
     ((plane_pl > 0) and (plane_p2 < 0))) then
   begin
        with polygon^ do
         begin
          delta_x := hidden_x_coor[p2]
                   - hidden_x_coor[pl];
          delta_y := hidden_y_coor[p2]
                   - hidden_y_coor[pl]; (* Since these *)
          delta_z := hidden_z_coor[p2]  (* values will be used more than *)
                   - hidden_z_coor[pl]; (* once, do them here *)
          delta_w := hidden_w_coor[p2]
                   - hidden_w_coor[pl];
         end;

(* calculate the parametric line equation constant T for the intersection *)
(* of the line Pl —>P2 and the split plane *)
       with splitpln^ do
         T_intersection := (-plane_pl)/((plane_a * delta_x) +
                                        (plane_b * delta_y) +
                                        (plane_c * delta_z));

       (* now calculate the x,y,z of the intersection point by *)
       (* substituting T into the parametric line equations for x,y,z *)

       with polygon^ do
         begin
          x_intersection := hidden_x_coor[pl] + (T_intersection * delta_x);
          y_intersection := hidden_y_coor[pl] + (T_intersection * delta_y);
          z_intersection := hidden_z_coor[pl] + (T_intersection * delta_z);
          w_intersection := hidden_w_coor[pl] + (T_intersection * delta_w);
```

```
(* A similar calculation can be performed to find the color values needed *)
(* this point.  Using T and knowing the values of say RED at P1 and P2 we *)
(* can calculate RED at the intersection vertex by :                      *)
(*                RED[p1] := RED[p1] + (T_intersection * (delta_red));    *)
(* This new value could then be placed into the vertex list for the new   *)
(* point                                                                  *)

          end;      (* with polygon^ *)
          (* place the intersection point in both vertex lists *)
         side_a := side_a + 1;
         side_b := side_b + 1;
         x_a[side_a] := x_intersection; x_b[side_b] := x_intersection;
         y_a[side_a] := y_intersection; y_b[side_b] := y_intersection;
         z_a[side_a] := z_intersection; z_b[side_b] := z_intersection;
         w_a[side_a] := w_intersection; w_b[side_b] := w_intersection;

     end;  (* test for intersection *)
(* see if output verticies make sense,  they must have at least 3 sides *)

 if ((side_a < 3) or (side_b < 3)) then
    writeln('poly_split: degenerate polygon created ')
 else
   placement;  (* put the vertex lists into the hidden linked list *)

end;  (* of split_poly *)

modend.
```

```
(**********************************************************************)
(*  Date: 4 Dec 83                                                    *)
(*  Version : 1.0                                                     *)
(*  Name: Sort Hidden                                                 *)
(*  Module Number: 5.1                                                *)
(*  Function:                                                         *)
(* This is the preprocessor to the Newell, Newell, and Sancha Hidden  *)
(* Surface Removal algorithm for the U of P CORE system.  It takes the *)
(* output primitives created from the current batch of updates and    *)
(* merges them into the linked list created by previous batches.  The *)
(* sorting is simply by z in viewing coordinates with the furthest z  *)
(* at the beginning of the list.  Nodes that have been moved are not  *)
(* used for comparision.                                              *)
(*  Inputs:   None.                                                   *)
(*  Outputs:  None.                                                   *)
(*  Global Variables Used:  Primitive and segment data structure      *)
(*                          Hidden surface linked list structure      *)
(*  Global Variables Changed: Hidden surface linked list structure    *)
(*  Global Tables Used: None.                                         *)
(*  Global Tables Changed: None.                                      *)
(*  Files Read:  None.                                                *)
(*  Files Written: None.                                              *)
(*  Modules Called:  Initialize hidden surface node, Delete hidden    *)
(*                   surface node                                     *)
(*  Calling Modules: End_Batch_of_Updates                           *)
(*                                                                    *)
(*  Author: Tom S. Wailes, Capt, USAF                                 *)
(*  History: This is the original module                              *)
(**********************************************************************)

Module SORT;

 const
   {$I defconst.src}
   (*EOC end of const section *)

 type
   {$I deftype.src}
   (*EOT end of type section *)

 var
   {$I extvar.src}
   (*EOV end of var section *)

 (*BOP beginning of procedures and functions *)
    EXTERNAL PROCEDURE DELETE_HIDDEN(L : LINKPTR; MODE : DISPOSER);
    EXTERNAL [3] PROCEDURE INIT_HIDDEN_LINK(PT : PRIM_PTR; MDE : ENTRY);
 (*EOP end of procedures and functions *)

procedure SORT_HIDDEN(start: SEG_PTR);

var
 seg : SEG_PTR;
 prim: PRIM_PTR;
 mergelist,next,present,link : LINKPTR;

begin

 mergelist := nil;    (* start new list of nodes to be added *)
 seg := start;        (* start search with first segment *)

 while seg <> nil do
    begin
      if seg^.visible then
         begin
           prim := seg^.prim_list;
           while prim <> nil do     (* examine all primitives for the seg *)
            begin
              if prim^.hidden = nil then
                 begin
                   new(link);                  (* get new hidden node *)
                   prim^.hidden := link;
                   init_hidden_link(prim,normal);  (* set it up *)
                   link^.forwrd := mergelist;
                   mergelist := link; (* place it into temp list *)
                 end
              else
                init_hidden_link(prim,update); (* recalculate image *)
             prim := prim^.next_prim;
           end;
        end
      else         (* segment not visible, if hidden node remove it! *)
        begin      (* this prevents unneeded computations!          *)
          prim := seg^.prim_list;  (* search all primitives *)
          while prim <> nil do
           begin
            if prim^.hidden <> nil then    (* remove them all *)
               delete_hidden(prim^.hidden,remove);
            prim := prim^.next_prim;
           end;
        end;
   seg := seg^.next_seg;   (* look at the next segment in the list *)
 end;

if mergelist <> nil then      (* check for merging entries *)
    begin
      present := hiddenlist;  (* set present to the furthest primitive *)
      if present = nil then   (* present will be nil on first hiddenlist *)
        begin
          hiddenlist := mergelist; (* place initial primitive in list *)
          present := mergelist;
          mergelist := mergelist^.forwrd;
          present^.forwrd := nil;   (* both back and forward pointers nil *)
        end;
      while mergelist <> nil do
        begin
```

```
          next := mergelist^.forwrd;      (* save pointer to next merging *)
                                          (* entry into the hidden structure *)


          (* because previous batching may have reordered the list *)
          (* this while ensures that the initial test will be either on *)
          (* primitive closest to the eye or on an unmoved primitive *)

          while ((present^.moved) and (present^.forwrd <> nil)) do
               present := present^.forwrd;

     if mergelist^.z_max_view > present^.z_max_view then  (* go left *)
              begin
                while ((( mergelist^.z_max_view > present^.z_max_view) or
                       (( mergelist^.z_max_view = present^.z_max_view) and
                        ( mergelist^.z_min_view > present^.z_min_view)) or
                       present^.moved) and ( present^.backwrd <> nil)) do

                        (* search left until proper place found, ignore *)
                        (* moved polygons, for they are out of order *)

                        present := present^.backwrd;  (* searches left *)

                        if ((mergelist^.z_max_view > present^.z_max_view) or
                          (( mergelist^.z_max_view = present^.z_max_view) and
                           ( mergelist^.z_min_view > present^.z_min_view)) or
                           present^.moved) then   (* merge at end of list *)
                           begin
                            present^.backwrd := mergelist; (* place left *)
                            mergelist^.forwrd := present;
                            hiddenlist := mergelist;(* this is the prim *)
                                                    (* furthest from eye *)
                         (* mergelist^.backwrd := nil   done in init *)
                           end
                         else
                           begin       (* normal insert *)
                            mergelist^.forwrd := present^.forwrd;
                             mergelist^.backwrd := present;
                             if present^.forwrd <> nil then
                                present^.forwrd^.backwrd := mergelist;
                             present^.forwrd := mergelist;
                           end;
             end
           else
             begin      (* search right *)
             while (((mergelist^.z_max_view < present^.z_max_view) or
                    (( mergelist^.z_max_view = present^.z_max_view) and
                     ( mergelist^.z_min_view < present^.z_min_view)) or
                       present^.moved) and (present^.forwrd <> nil)) do
                           present := present^.forwrd;
              if ((mergelist^.z_max_view < present^.z_max_view) or
                (( mergelist^.z_max_view = present^.z_max_view) and
                 ( mergelist^.z_min_view < present^.z_min_view)) or
                  present^.moved) then
                  begin
                    present^.forwrd := mergelist; (* place on right *)
                    mergelist^.backwrd := present;
                    mergelist^.forwrd := nil;
                  end
             else
               begin          (* normal insert *)
               mergelist^.forwrd := present;
                 mergelist^.backwrd := present^.backwrd;
                 if present^.backwrd <> nil then
                    present^.backwrd^.forwrd := mergelist;
                 present^.backwrd:= mergelist;
               end;
           end;
      mergelist := next;     (* get the next entry entering the structure *)
     end;
   end;
 end;
 modend.
```

```
(***********************************************************************)
(*  Date: 4 Dec 83                                                     *)
(*  Version : 1.0                                                      *)
(*  Name: Initialize Hidden Surface Node                               *)
(*  Module Number: 5.1.1                                               *)
(*  Function:                                                          *)
(* This is the hidden surface node initialization routine.  It takes*)
(* a pointer to an output primitive and prepares it for use in the  *)
(* Newell, Newell, and Sancha hidden surface algorithm within the   *)
(* U of P CORE subroutine package.  Four possible cases exist for   *)
(* primitives to be set.  On the batch of updates in  which a       *)
(* primitive is created, this routine completely sets all of the    *)
(* parameters to their initial value( this is called a normal entry)*)
(* On the batches of updates that follow, the pointers have been    *)
(* set placing the primitive in the correct ordering for the last   *)
(* picture, and because the present picture should have only small  *)
(* changes, we do not reset them( this is called an update entry).  *)
(* Both of these types of entries require us to recalculate the     *)
(* image space coordinates of the primitive`s points.  If a polygon *)
(* is split during the application of the hidden surface removal    *)
(* algorithm the resolver procedure has already placed the image    *)
(* transformed coordinates into the hidden surface the incoming     *)
(* hidden surface node.  The original polygon is initialized with   *)
(* an entry of split and the new hidden node is entered with        *)
(* a value of new_prim.  The difference here is that the original   *)
(* polygon`s pointers are valid and the new node's pointers must    *)
(* be set to nil.  The min and max in screen x and y, the min and   *)
(* max z in viewing coordinates, and the plane equations for the    *)
(* primitive must be calculated for all cases.                      *)
(*  Inputs:  P (pointer to primitive structure)                      *)
(*           mode (flag to tell what kind of update to perform)      *)
(*  Outputs: None.                                                   *)
(*  Global Variables Used:  Primitive and Hidden surface Data        *)
(*                          Structures                               *)
(*                          View_State projection type              *)
(*  Global Variables Changed: Hidden surface structure               *)
(*  Global Tables Used: None.                                        *)
(*  Global Tables Changed: None.                                     *)
(*  Files Read:  None.                                               *)
(*  Files Written:  None.                                            *)
(*  Modules Called:  Make Matrix (create 4X4 image transformation    *)
(*                                matrix)                            *)
(*                   Image transformation (does image transformation*)
(*  Calling Modules: End_Batch_of_Updates                         *)
(*                                                                   *)
(*  Author: Tom S. Wailes, Capt, USAF                                *)
(*  History: This is the original module                             *)
(***********************************************************************)

Module H0_INITHID;

 const
```

```
  {$I defconst.src}
  (*EOC end of const section *)

type
  {$I deftype.src}
  (*EOT end of type section *)

var
  {$I extvar.src}
  (*EOV end of var section *)

(*BOP beginning of procedures and functions *)
  {$I DDUTLEXT.SRC} (* concol, scr<->ndc, image xforms *)
(*EOP end of procedures and functions *)


procedure INIT_HIDDEN_LINK(p:PRIM_PTR; mode : ENTRY);

var
 i,j,number_of_points : integer;
 x,y : real;

(****************************************************************************)
(*    main routine                                                          *)
(****************************************************************************)

begin
 with p^.hidden^ do
   begin
     number_of_points := p^.coorcnt;
     case mode of
      normal:
        begin
          next_id := next_id + 1;
          output_primitive := p;    (* set parent pointer *)
          forwrd := nil;            (* initially out of list *)
          backwrd := nil;
          moved := false;           (* initially placed in order *)
          makemat(p^.owner_seg^.image_trans);    (* perform image transform *)
          for i:=1 to number_of_points do
              imgtrans(p^.xcoor[i],p^.ycoor[i],p^.zcoor[i],p^.wcoor[i],
                 hidden_x_coor[i],hidden_y_coor[i],hidden_z_coor[i],
                 hidden_w_coor[i]);

        end;
      update:
        begin
          makemat(p^.owner_seg^.image_trans);    (* perform image transform *)
          for i:=1 to number_of_points do
              imgtrans(p^.xcoor[i],p^.ycoor[i],p^.zcoor[i],p^.wcoor[i],
                 hidden_x_coor[i],hidden_y_coor[i],hidden_z_coor[i],
                 hidden_w_coor[i]);

       end;
```

```
        new_prim:
          begin
            next_id := next_id + 1;
            output_primitive := p;    (* set parent pointer *)
            forwrd := nil;            (* initially out of list *)
            backwrd := nil;
            moved := false;           (* initially placed in order *)
          end;
       (* split: nothing; *)
    end; (* case *)

  enter := mode;       (* save type of entry *)


  (* calculate the x,y screen extents of the transformed points *)

  if view_state.pjtyp = 2 then   (* perspective case *)
   begin
     x_min_screen := hidden_x_coor[1]/hidden_w_coor[1]; (* screen position *)
     y_min_screen := hidden_y_coor[1]/hidden_w_coor[1]; (* depends on       *)
    end                                                 (* homogeneous coor  *)
  else
    begin
      x_min_screen := hidden_x_coor[1];     (* parallel case does not depend *)
      y_min_screen := hidden_y_coor[1];     (* on homogeneous coor    *)
    end;
  z_min_view   := hidden_z_coor[1];
  x_max_screen := x_min_screen;
  y_max_screen := y_min_screen;
  z_max_view   := z_min_view;

  for i:= 2 to number_of_points do        (* test each point and update the *)
   begin                                 (* min and max as appropriate     *)

  if view_state.pjtyp = 2 then   (* perspective case *)
   begin
     x := hidden_x_coor[i]/hidden_w_coor[i]; (* screen position *)
     y := hidden_y_coor[i]/hidden_w_coor[i]; (* depends on      *)
    end                                      (* homogeneous coor   *)
  else
    begin
      x := hidden_x_coor[i];     (* parallel case does not depend *)
      y := hidden_y_coor[i];     (* on homogeneous coor    *)
    end;
     if x_min_screen > x then
        x_min_screen := x
     else if x_max_screen < x then
             x_max_screen := x;
     if y_min_screen > y then
        y_min_screen := y
     else if y_max_screen < y then
             y_max_screen := y;
     if z_min_view > hidden_z_coor[i] then
        z_min_view := hidden_z_coor[i]
```

```
 else if z_max_view <  hidden_z_coor[i] then
          z_max_view := hidden_z_coor[i];
end;

(* calculate the plane equations coefficients from a method found in *)
(* Sutherland, I.E., R.F. Sproull, and R.A. Schumacker, " A Characteriza- *)
(* tion of Ten Hidden-Surface Algorithms," Computing Surveys,  *)
(* 6(1) : 14-15(Mar 74)    *)
 i:=1;
 j:=2;
 plane_a := 0;
 plane_b := 0;
 plane_c := 0;

 repeat
   plane_a := plane_a + ((hidden_y_coor[i] - hidden_y_coor[j])
                       * (hidden_z_coor[i] + hidden_z_coor[j]));

   plane_b := plane_b + ((hidden_z_coor[i] - hidden_z_coor[j])
                       * (hidden_x_coor[i] + hidden_x_coor[j]));

   plane_c := plane_c + ((hidden_x_coor[i] - hidden_x_coor[j])
                       * (hidden_y_coor[i] + hidden_y_coor[j]));

   i := i+1;
   j := j+1;

   if i = number_of_points then
      j := 1;

 until i > number_of_points;

(* to find d simply place the first point into the plane equation and solve *)

 plane_d := -(plane_a * hidden_x_coor[1]) - (plane_b * hidden_y_coor[1])
          - (plane_c * hidden_z_coor[1]);

 if plane_d < 0.0 then  (* user has defined the object counter-clockwise *)
   begin
    plane_a := -plane_a;
    plane_b := -plane_b;
    plane_c := -plane_c;
    plane_d := -plane_d;
   end;
end; (* with p^.hidden *)
end; (* init_hidden link *)

modend.
```

```
(***************************************************************************)
(*  Date: 4 Dec 83                                                        *)
(*  Version : 1.0                                                         *)
(*  Name: Delete Hidden Surface Node                                      *)
(*  Module Number: 5.1.2                                                  *)
(*  Function:                                                             *)
(* This procedure is used to delete nodes from the hidden surface linked *)
(* list produced during the operation of the Newell, Newell, and Sancha  *)
(* Algorithm.  Two basic modes of operation are controlled by the input  *)
(* mode.  If the mode is retain,  then the node is to be removed from    *)
(* the linked list but not disposed. Otherwise if the user specifies     *)
(* remove, then the node will be deleted from the linked list and its    *)
(* memory space given back to the operating system by a call to dispose. *)
(*  Inputs:  L (pointer to node to be deleted), mode (type of deletion)  *)
(*  Outputs: None.                                                        *)
(*  Global Variables Used:  Hidden surface data structure                 *)
(*  Global Variables Changed:  Hidden surface data structure              *)
(*  Global Tables Used: None.                                             *)
(*  Global Tables Changed:   None.                                        *)
(*  Files Read: None.                                                     *)
(*  Files Written:   None.                                                *)
(*  Modules Called:  None.                                                *)
(*  Calling Modules:  DDEMPTYSEG, Sorter                                  *)
(*                                                                        *)
(*  Author: Tom S. Wailes, Capt, USAF                                     *)
(*  History: This is the original module                                  *)
(***************************************************************************)


Module delhid;
(* Modified for VAX by SMP 7/82 *)
(*   (including auto-generation of CVTP
   sections, which is why some may be empty. *)

 const
   [$I defconst.src]
   (*EOC end of const section *)

 type
   [$I deftype.src]
   (*EOT end of type section *)

 var
   [$I extvar.src]
   (*EOV end of var section *)

procedure DELETE_HIDDEN(L : LINKPTR; mode : DISPOSER);

begin
  (* hiddenlist points to the farthest primitive from the eye *)


  if hiddenlist = L  then    (* if L is the furthest primitive then *)
```

```
      if L^.backwrd <> nil then  (* see if L is the only primitive *)
         begin
           hiddenlist := L^.backwrd;   (* if it is not then move *)
           L^.backwrd^.forwrd := nil; (* the pointers to next to last*)
         end
      else
        hiddenlist := nil     (* we are removing the last primitive *)
   else  (* not the last primitive *)
      begin
       L^.forwrd^.backwrd := L^.backwrd;
       if L^.backwrd <> nil then  (* is this the closest primitive to eye *)
          L^.backwrd^.forwrd := L^.forwrd;   (* no *)
      end;

   (* now reclaim memory if possible *)

  if mode = remove then
     begin
       L^.output_primitive^.hidden := nil;
       dispose(L);
     end;

end; (* delete hidden *)


modend.
```

Appendix B

Maintaining the Flight Dynamics Laboratory CORE Graphics System

The CORE graphics system developed to support the Flight Dynamics Laboratory can be thought of as a collection of many individual subroutine modules grouped by major function into four overlay libraries. These overlay libraries are connected to the users main program by LINKMT as the last step before execution. This appendix will give specific guidance in the steps necessary to create a graphics program that uses the CORE and additionally provide documentation on how to modify the existing CORE subroutines.

First, the basic structure of a program created under MT+86 is explained including a discussion of the various limits in program, data, stack, and heap space caused by the Pascal MT+86 system. Then the LINKMT link line commands needed to "link" a user program to the CORE system are described. Finally, the steps needed to modify the CORE libraries are explained including how to use the undocumented system utility program called STRIP.

Program Structure

Programs created by Pascal MT+86 are divided into four memory areas called segments (Figure 10). The first segment, called the program segment, contains the actual instructions that are executed by the 8086. A second segment contains the data manipulated during execution and is therefore called the data segment. Thirdly, there is an extra segment that is used by Pascal MT+86 for the Pascal "heap" space. Finally, because of the trend toward modular program development requiring the passing of information between many different procedures on a "stack", Intel defines a stack segment.

Addressing on the 8086 is performed by adding 16 bit offset values to four internal 16 bit registers (one for each of the program, data, stack, and extra segments). The use of these registers reduces the amount of bits needed by instructions to specify an address, but under the present compact memory model of PASCAL MT+86, also limits the address space of program, data, and stack segments to 64K bytes (fortunately you can use FULLHEAP in the link line to allow up to 1 megabyte of heap (extra segment) storage area). Although undocumented, the compiler apparently can not generate the necessary segment register modification instructions needed to allow larger program, data, and stack memory areas. Therefore, to overcome program and data area size limitations, Pascal MT+86 allows the user to define program and data area overlays.

An overlay area is simply a part of the 64K program and data areas that is set aside by the linker to be shared by overlay modules (groups of procedures). Although Pascal MT+86 allows the user to define several places in memory as overlay areas, the present CORE system uses only one 32K overlay area. During execution, an overlay manager subroutine will be called whenever a procedure assigned to this overlay area is required by the main program. The overlay manager will read the procedure's overlay module into memory, find its address by searching a symbol table found near the end of the overlay module, and then branch to that address. Placement of a procedure within an overlay is done by inserting an overlay module number within Pascal MT+86 external procedure declarations referencing the procedure, and by placing the compiled procedure inside an overlay module library.

Creating an overlay library requires first making a .BLD file that contains a list (one procedure name per line) of all the .R86 (compiled procedure) files to be placed within the library. After this file is created and all of the procedures within the overlay have been compiled, LIBMT is used to create the library. LIBMT will concatenate all of the files listed in the .BLD file into one large .R86 file that will be used in linking the overlay modules (note that the present .BLD files require that all compiled .R86 files are on drive C!). The .BLD library files needed for the present three overlay modules are OVLY1,OVLY2,and OVLY3 (e.g. a sample command line for the overlay 1 library is: LIBMT OVLY1).

Overlay one is rather small and contains the procedures used one time only for initialization and termination. This frees the two main overlays from the burden of infrequently called procedures and gives them more room for development and debugging. Overlay two contains most of the user callable CORE procedures that set viewing parameters, draw lines, draw polygons etc. along with the utility procedures needed to support them such as the graphics "clipping" routine. Finally, overlay three contains the device dependent device driver routines along with the hidden surface routines developed during this research.

Program segment space is divided into four separate areas: the user's main program, the Pascal support procedures, a gap between the support procedures and the overlay area, and the overlay area itself. When the main program is linked, the main program will be loaded into the low address portion of the program segment. The Pascal support procedures are loaded directly after the main program and as a rule take more room than the main program. The size of the combined memory area of the first two sections can be found by doing a preliminary link of the main program. The number following the printout "TOTAL CODE:" is this size. Because the linker will load the program segment starting at offset address 0000 (hex), this is also the offset address of the end of the combined program and support procedure area.

An 80 byte gap must exist between the end of the main code section and the start of the overlay area. Larger gaps are acceptable and therefore the present parameter settings use an address that will both allow more than 80 bytes of area between the program and overlay area and allow room at the upper end of memory for expansion of the overlays (useful when adding debug statements). Specifying the beginning of the overlay area is done using the V1 parameter of the link line when linking the main program and by using the P parameter when linking the overlays (be SURE to use the same number in both places!). An upper bound to this overlay area is set using the R parameter in the main program link line.

The data area specifications are done in a similar fashion when overlay procedures have their own separate data areas. However, in this implementation the main program "owns" all of the global variables needed by the overlays, and the overlays use only temporary variables during their execution. Thus, after performing a preliminary main program link, the number following "TOTAL DATA:" is used as the D parameter on subsequent main program links.

Finally, it should be stated that there are a few routines that are needed by more than one overlay group such as the matrix multiplication routine. These few routines are placed into the "root" or main program by a subroutine that is added to the end of the global variable include file to create the compiler references for them. The library that contains these routines (ROOT) is not organized as an overlay but instead is joined to the main program by the main program link.

The Present LINKMT Command Lines

Linking of the CORE to a user program requires first linking the main program and then linking the three overlays. The main program link will create a special .SYM symbol file that will be used by the linker when linking the overlays. Therefore, the main procedure must be linked first. Of course, if the main program is changed, then both it and all overlays must be relinked. However, because linking is done by symbol, changes to the overlays do not require the subsequent relinking of the main program, only the relinking of overlays that are changed.

The present link line for linking the main program extends further than the margins for this report, therefore consider this line as one continuous string of characters entered on the system console. To link the main program enter:

```
LINKMT B:TESTPRG,B:ROOT/S,TRANCEND/S,FULLHEAP/S,87REALS/S,
PASLIB/S/D:2000/V1:7000/R:FFEF/X:3000/Z:1000/L
```

LINKMT = Name of the MT+86 linker program.

B:TESTPRG = Substitute the name of the main program here.

B:ROOT = Library containing the procedures used by more than one overlay.

/S = This library is searchable, so only procedures that were referenced will be included when linking.

TRANCEND = Pascal Mathematics subroutine package.

FULLHEAP = Pascal Heap space mangement package.

87REALS = 8087 co-processor subroutines.

PASLIB = Main Pascal Library.

D:2000 = Set size of main program data area at 2000.

V1:7000 = Set overlay area 1 (we only use 1) beginning address to 7000. Overlay module numbers 1-16 will be placed in this area. Since we have 3 overlays numbered 1-3, they will all start at this address.

R:FFEF = Set upper bound for overlays at FFEF (according to Digital, positively the highest possible address for overlays).

Z:1000 = Set the max size of the stack segment at 64K (max size possible).

x:3000 = Set the maximum size of the heap space at 192K (maximum value is FFFF or 1 megabyte).

L = print out the procedure offset addresses as they are linked (a good reference later when trying to find out the size of the program area vs the system routines.)

The current Overlay link line is:

```
LINKMT B:TESTPRG.001=B:TESTPRG/O:1,B:OVLY1,PASLIB/S/P:7000/L
```

LINKMT = The Pascal MT+86 linker program.

B:TESTPRG.001 = The name of the file created during the link, in this case the file created will be the first overlay module for TESTPRG.

B:TESTPRG/O:1 = This file name signals to the linker that the file to be created is going to be an overlay and that the .SYM file to be used for linking is named TESTPRG.SYM.

The 1 indicates that this overlay will be loaded into the area specified by the V1 parameter of the main link. For our case this number could be as high as 16 without changing the loading, however, for consistancy I use the overlay module number for this parameter.

B:OVLY1 = This is the name of the overlay library to use when creating the overlay. Obviously, when creating overlay module number 2 this file name would be B:OVLY2.

PASLIB/S = Satisfy any system subroutines not called by the main program using PASLIB.

P:7000 = The beginning address of the overlay. This always agrees with the V1 parameter of the main program link line.

L = Print out the overlay procedure offset addresses (useful when determining if an overlay has exceeded its maximum size).

Using the above parameters will result in a system that resides in memory as:

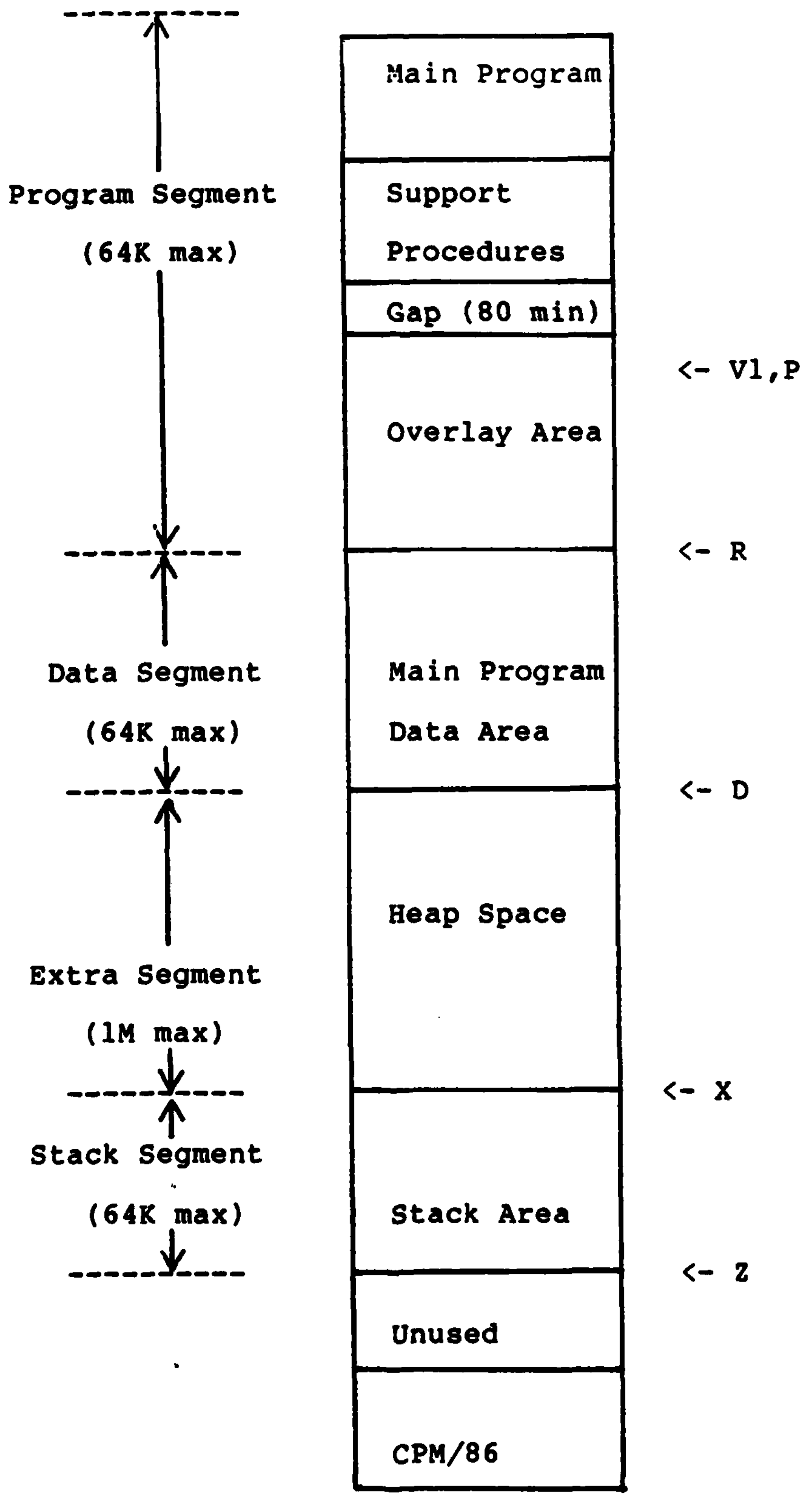


Figure 10. Memory Layout of CORE System

Modifing the Overlays

Modification of the overlays is accomplished by simply changing the procedures desired, compiling the new procedures, creating new module libraries using LIBMT, and finally relinking the new overlays to a root program segment. This is the only processing involved as long as major additions to the overlay area of the program segment are avoided. However, memory conflicts arise when the overlay procedures become too large for their assigned space.

Whenever an overlay exceeds the space set aside by the linker by the values of V1,P and R parameters, problems develop when the overlay manager tries to place the overlay into its assigned position. Located directly behind the procedures within an overlay module is a symbol table that will be used to find the address of procedures contained within the overlay. In addition, this table is followed by a record that contains from 1 to 4 thousand zeros used when performing the overlay linkage. The overlay manager first fills the area specified by the V1,P and R parameters. If there are more records in the overlay file, the overlay manager will overwrite the last record successfully transferred into the overlay area by the remaining records in the overlay file until the end of the overlay file is reached. Thus, one of the first things that will be destroyed whenever an overlay becomes too large for the overlay space will be the symbol table used to find procedure addresses. When this occurs the overlay manager will printout a fatal error message stating that it was not able to find the requested procedure name within the overlay it was assigned. This type of error can be confirmed by looking at the linkage offset addresses for the overlay module in which this error occurs. If the address of the last procedure linked is over F200 (assuming R with a value of FFEF) then it is likely that symbol table overwrite is occuring.

When overwriting is suspected there are four basic options available to solve the problem. First and easiest, the user may be able to remove some debugging or other unneccessary code from the overlay to reduce its size. The V1,P and parameters may be able to be changed (reduced in address) to create more room, but this means less area available for the user program. Another alternative is to remove a procedure from the overlay, which usually involves some loss of program function. Finally, the user can both reduce the size of the symbol table found at the end of the overlay module and eliminate the table of zeros completely by using the undocumented system utility program called STRIP.

Using STRIP

To use STRIP, the user must have two things. First he must know the beginning address of the overlay that he wishes to reduce in size, and second he must have a list of all the procedures within the overlay in the order in which they were linked. The first information is simply the P parameter used during the linking of the overlay. Finding the second bit of information is more involved.

To obtain this list of procedures in the proper order the user should perform three steps, in sequence. First the user should relink the overlay adding /M to the link line to get a load map of the link. This map will list the names of every module, procedure, and global variable used by the overlay in the proper order for strip. However, we wish to keep only the procedure names needed by the overlay manager, so some method is needed to eliminate the extra names.

The module names can be eliminated by striking thru the names on the map that contain an alphnumeric prefix consisting of a letter followed by 2 digits (e.g. A00INIT). These module names were originally created during the Pascal code conversion process to make them unique. Variable names are also easy to spot by examining the names in the global variable include file and the variable declaration section of the user's main program. Unfortunately, some of the names left are due to Pascal system support procedures which also must be eliminated. These require the use of DDT86.

Prior to using DDT86, it is best to obtain the starting address of the symbol table of the overlay. The easiest method is to use the first part of the STRIP utility. To get this address simply type STRIP at the operators console. STRIP will prompt you for the name of the overlay. After typing the name in (e.g. B:TESTPRG.003), STRIP will prompt you for the starting address of the overlay. Type in the P parameter (e.g. 7000) and STRIP will print out the address of the symbol table. Exit STRIP by typing control C.

The next step requires access to the general purpose debugging tool named DDT86. Because of its potential destructive power in the hands of an unfamiliar user, it is protected on the Flight Dynamics system by a password. After getting into DDT86 using the password, two commands will be issued. To read the overlay into memory so that we can examine the symbol table use the DDT "R" (read) command (e.g. R B:TESTPRG.003). This will place the overlay in memory for our examination. Next, use the "D" (display) command to print out the symbol table in ASCII using the beginning address of the symbol table (obtained using STRIP above) as the address to start the display (e.g. D8754). Compare the remaining link map entries against the names in the symbol table. Strike out any names that are not found in the symbol table. Because of the power of DDT86, leave this step in the hands of an experienced user of the system.

At this point, the modified link map contains just the names of the procedures in the overlay. As long as no procedures are added to or subtracted from the overlay, this listing of names will remain valid (even if modifications are done to the procedures within the overlay).

Now it is time to actually perform the stripping operation. First enter STRIP as before by typing in STRIP, the overlay file name, and the starting address of the overlay. STRIP will respond with the prompt "ENTER NEXT NAME TO KEEP". Enter the first name on the modified map listing. STRIP will respond with "DO YOU WISH TO KEEP <name>?". Enter "Y" and STRIP will again prompt with "ENTER NEXT NAME TO KEEP". Enter the next name in the list, followed by "Y" etc. until the list of names is exhausted. If you misspell a symbol table entry, enter a name out of order, or if you type too many characters (accidentially insert a blank) then strip will not find the name in the symbol table and will terminate asking if you wish to keep the now incorrect symbol table. If you make any of these errors, type "Q" and then start STRIP over again.

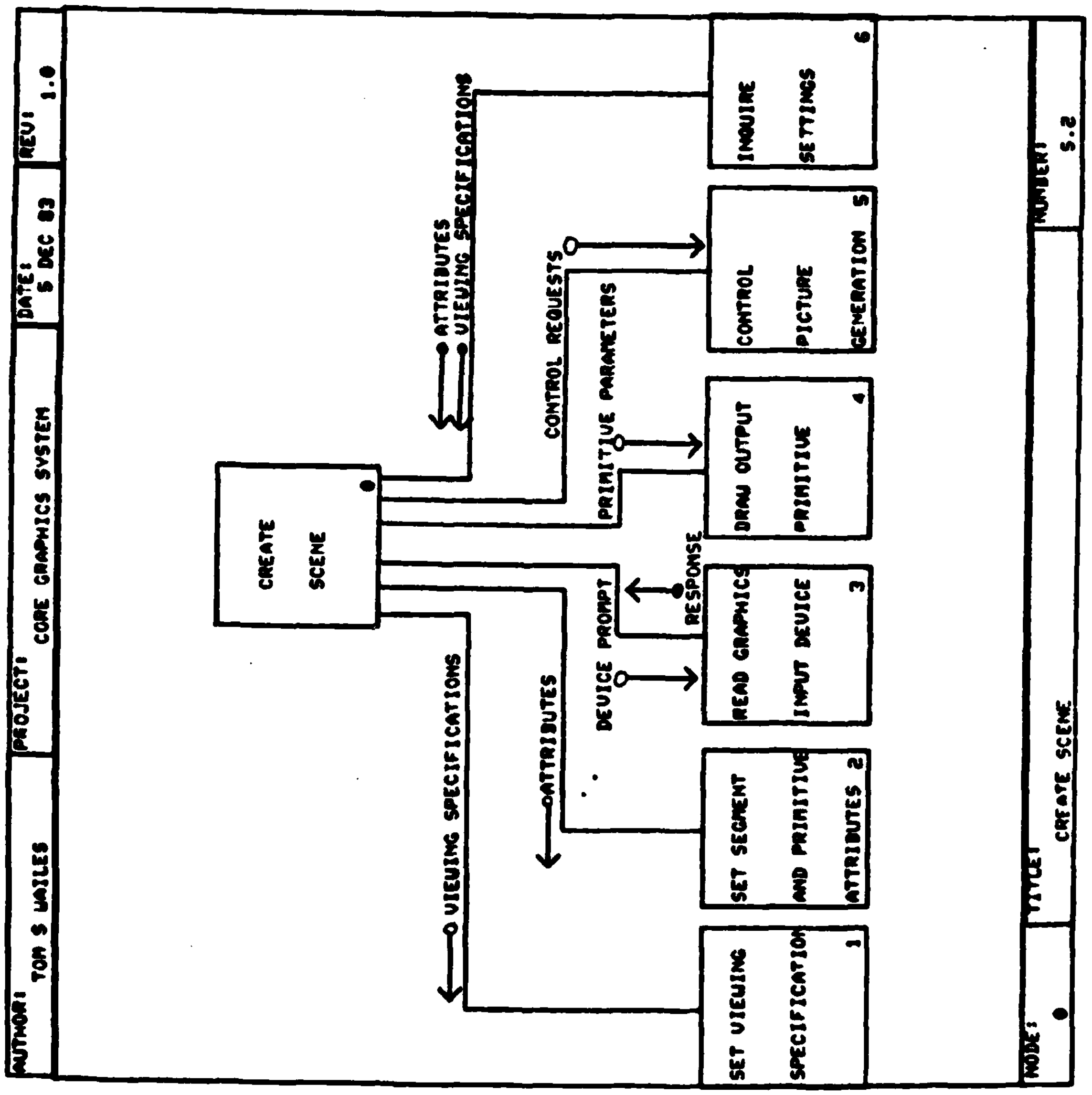
Once the user makes it completely through the list, he enters a blank followed by a carriage return to force the STRIP program to terminate (actually any incorrect name will do but a blank is easier). STRIP will then printout the newly created symbol table and prompt the user with "DO YOU WISH TO KEEP?". To keep this new file the user must enter exactly "YESDOIT" without the quotes (no other input will work!). STRIP will then replace the original overlay with a file that has a reduced symbol table and additionally is free from the record of zeros.

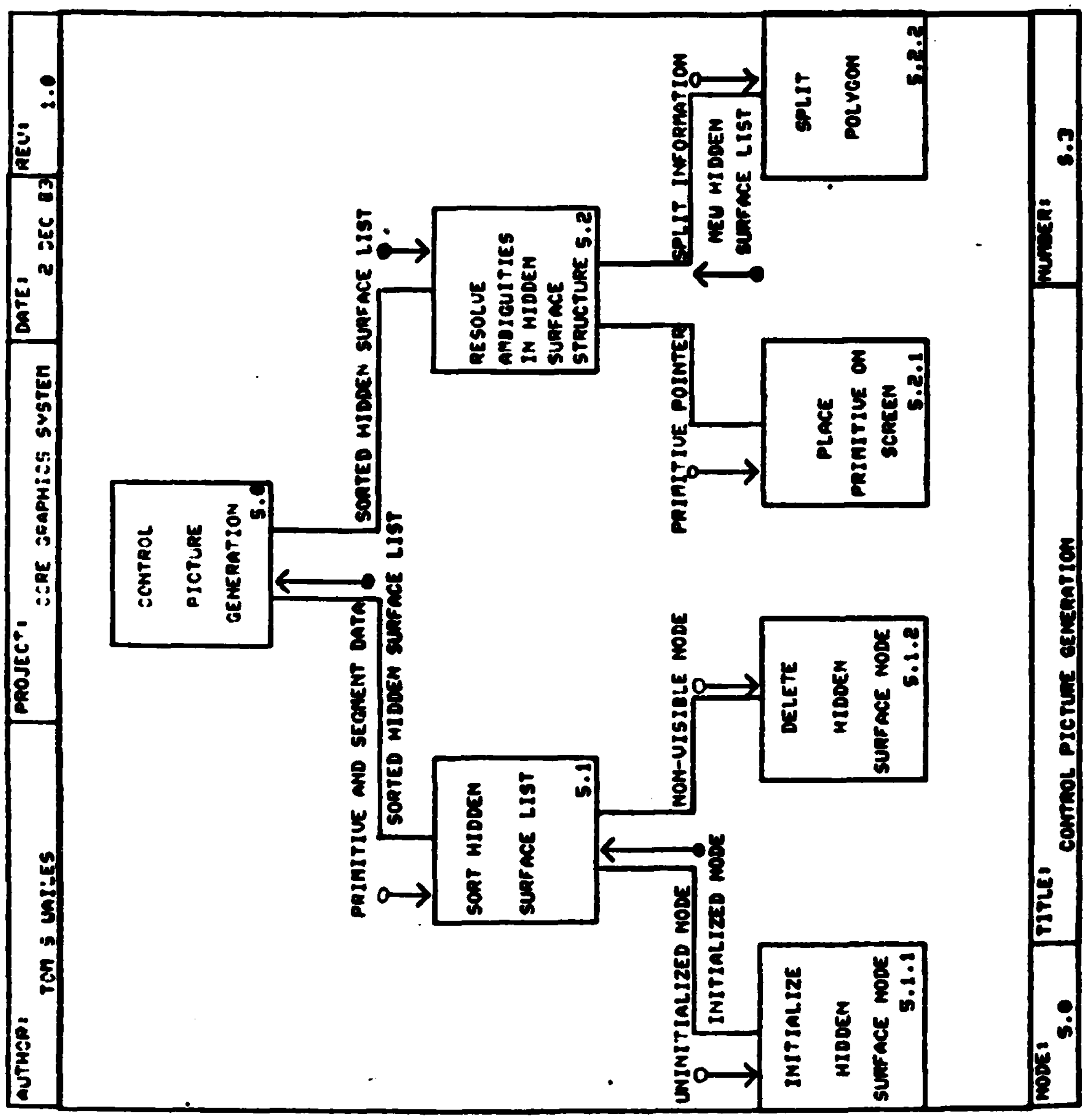
This appendix has discussed the creation and maintainance of the three CORE overlay areas. For answers to additional questions, refer to the Pascal MT+86 Language Reference Manual.

Appendix C:

Hidden Surface Interface to Main CORE Package

This appendix shows the interface between the main CORE subroutines and the hidden surface subroutines. The first structure chart shows an overall view of the CORE, with a Create Scene process that represents the user's program. The other six processes are the six main groupings of Subroutine Functions within the CORE.




AUTHOR: TOM S WAILES
PROJECT: CORE GRAPHICS SYSTEM
DATE: 2 DEC 83
REV: 1.0
CONTROL PICTURE GENERATION 5.0
SORT HIDDEN SURFACE LIST 5.1
RESOLVE AMBIGUITIES IN HIDDEN SURFACE STRUCTURE 5.2
INITIALIZE HIDDEN SURFACE NODE 5.1.1
DELETE HIDDEN SURFACE NODE 5.1.2
PLACE PRIMITIVE ON SCREEN 5.2.1
SPLIT POLYGON 5.2.2
PRIMITIVE AND SEGMENT DATA
SORTED HIDDEN SURFACE LIST
SORTED HIDDEN SURFACE LIST
SORTED HIDDEN SURFACE LIST
UNINITIALIZED NODE
INITIALIZED NODE
INITIALIZED NODE
NON-VISIBLE NODE
PRIMITIVE POINTER
SPLIT INFORMATION
NEW HIDDEN SURFACE LIST
NODE: 5.0
TITLE: CONTROL PICTURE GENERATION
NUMBER: 5.3

This Structure Chart shows the interface between the hidden surface routines and the main CORE package.

Vita

Tom Stotts Wailes was born June 29 1955 in Shreveport, Louisiana. He graduated from First Baptist High School of Shreveport in 1973, and attended the Air Force Academy from which he received the degree of Bachelor of Science in Computer Science in 1977. Upon graduation, he worked for 5 months as a test manager for the GBU-15, a guided glide bomb. He then attended pilot training at Reese AFB, Lubbock Texas and was unfortunately medically disqualified after soloing in the T-37 trainer aircraft. He then served as a programmer, systems analyst, and supervisor in the SAGE Programming Agency, Luke AFB, Glendale Arizona until entering the School of Engineering, Air Force Institute of Technology, in May 1982.

Permanent Address: Rt 5 Box 129A
Coushatta, La
71019

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

AD-A138239

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; Distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/MA/83D-9	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering	6b. OFFICE SYMBOL (If applicable) AFIT/ENC
7a. NAME OF MONITORING ORGANIZATION	
6c. ADDRESS (City, State and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB, Ohio 45433	7b. ADDRESS (City, State and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Flight Dynamics Laboratory	8b. OFFICE SYMBOL (If applicable) AFWAL/FIGR
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (City, State and ZIP Code) Air Force Wright Aeronautical Laboratory Wright-Patterson AFB, Ohio 45433	10. SOURCE OF FUNDING NOS.
11. TITLE (Include Security Classification) See Box 19	

10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.

12. PERSONAL AUTHOR(S)
Tom S. Wailes, B.S., Capt, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
MS Thesis	FROM ________ TO ________	1983 December 7	163

16. SUPPLEMENTARY NOTATION

Approved for public release: IAW AFR 190-17
[illegible]
LYNN E. WOLAVER 7 Feb 84
Dean for Research and Professional Development
[illegible]
Wright-Patterson AFB OH 45433

17. COSATI CODES

FIELD	GROUP	SUB. GR.
09	02	
12	01	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Computer Graphics, Microprocessors, Algoritlms

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Title: Placing Hidden Surface Removal Within the CORE
Graphics Standard: An Example

Thesis Chairman: Charles W. Richard Jr.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
UNCLASSIFIED/UNLIMITED [✓] SAME AS RPT. ☐ DTIC USERS ☐

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
Charles W. Richard Jr.	513-255-3098	AFIT/ENC

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

Block 19 (continued):

Abstract:

A pascal based ACM proposed standard (CORE) graphics system from the University of Pennsylvania was converted to run on an 8086/8087 microcomputer. Additionally, a non-restrictive full 3-D hidden surface removal algorithm for surfaces modeled by opaque polygons was implemented. This algorithm is an extension of the list priority algorithm created by Newell, Newell, and Sancha. By saving the priority list created by the algorithm from one batch of updates to the next, computational savings are possible. Although the present algorithm can only be used on raster type display surfaces, this algorithm could be used as a prelude to other scan line hidden surface removal algorithms to provide support for vector type displays.

END

FILMED

3-84

DTIC